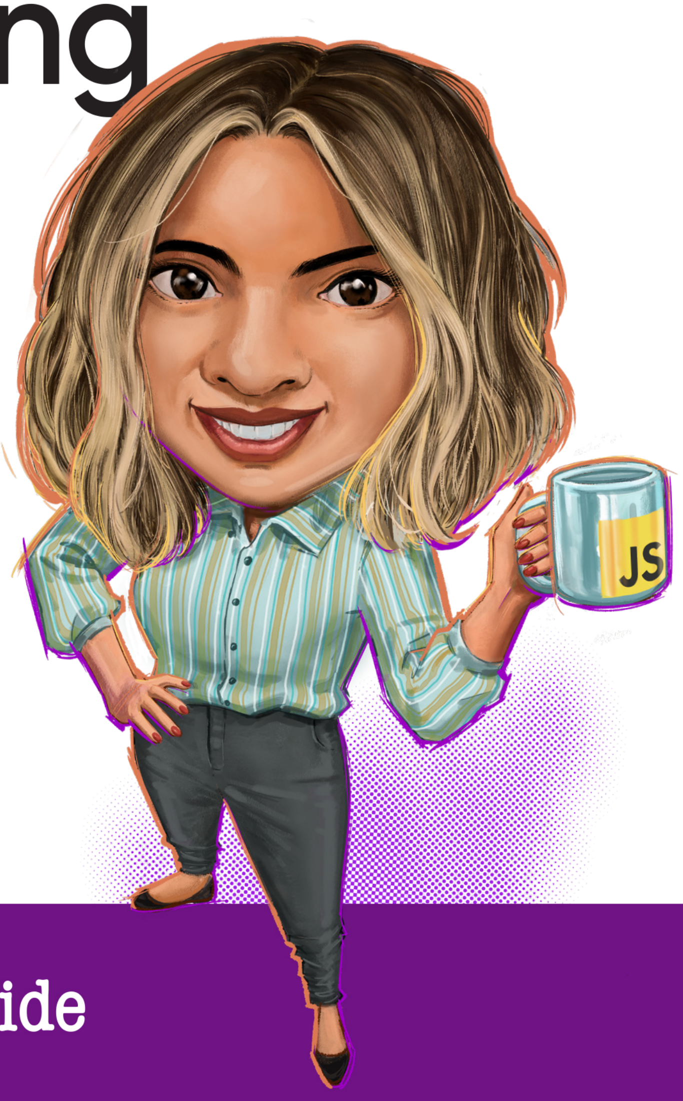
O'REILLY®

Second Edition

Head First JavaScript Programming

A Learner's Guide to Modern JavaScript

Eric Freeman &
Elisabeth Robson

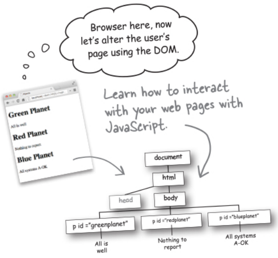


A Brain-Friendly Guide

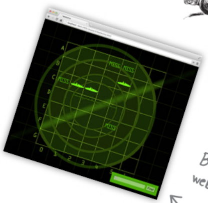
Head First
JavaScript Programming

What will you learn from this book?

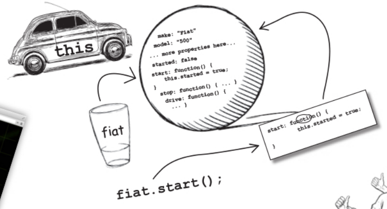
The new edition of this brain-friendly guide takes you through a comprehensive journey into modern JavaScript, covering everything from core language fundamentals to today's cutting-edge features. You'll dive into the nuances of JavaScript types and the unparalleled flexibility of its functions. You'll also learn how to expertly navigate classes and objects, and finally understand closures. But that's just the beginning. You'll also get hands-on with the browser's document object model (DOM), engaging with JavaScript in exciting ways. You won't just be reading—you'll be playing games, solving puzzles, pondering mysteries, and interacting with JavaScript as never before. And you'll write real code, lots of it, so you can start building your own applications.



Learn how to interact with your web pages with JavaScript.



Build real, interactive web applications.



```
make: "Fiat"
model: "500"
... more properties here...
started: false
start: function() {
  this.started = true;
}
stop: function() { ... }
drive: function() {
  ... }
```

```
start: function() {
         this.started = true;
}
```

```
fiat.start();
```

Work through real-world problem solving to know how to best approach writing your code.

What's so special about this book?

If you've read a Head First book, you know what to expect: a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. With this book, you'll learn JavaScript through a multisensory experience that engages your mind—rather than a text-heavy approach that puts you to sleep.

> "This book unlocks the power of JavaScript in a way that's never been done before! Say goodbye to boring code tutorials and hello to a fun, innovative way to master one of the most popular programming languages out there. Get ready to revolutionize the way you learn JavaScript!"
>
> **—Doreen Lorenzo**
> Assistant Dean, School of Design
> and Creative Technologies,
> University of Texas at Austin

> "Freeman and Robson are two of the best teachers around, and no one knows JavaScript better than they do. This is one of those rare books about software and computing that I can recommend without reservation."
>
> **—David Gelernter**
> Professor of Computer Science,
> Yale University

> "The clear, engaging, conversational tone of this book, coupled with accessible examples, exercises, and reader challenges, results in an indispensable tool for motivated, independent learners."
>
> **—Josh Sharfman**
> Shalhevet High School, Computer Science

JAVASCRIPT

US $69.99 **CAN $87.99**
ISBN: 978-1-098-14794-5

O'REILLY®

Praise for *Head First JavaScript Programming*

"This book unlocks the power of JavaScript in a way that's never been done before! It engages the reader with a dynamic experience. Say goodbye to boring code tutorials and hello to a fun, innovative way to master one of the most popular programming languages out there. Get ready to revolutionize the way you learn JavaScript!"

— Doreen Lorenzo, Assistant Dean, School of Design and Creative Technologies, University of Texas at Austin

"JavaScript is fundamental to the Web; if JavaScript were somehow to disappear, the Web as we know it would be gone. Everyone who is interested in Web software must know JavaScript–and, naturally, there are a million ways to learn it. But if I had to pick the best way, that would be easy: this book. The authors have known JavaScript inside-out since it was born. (In fact Java itself was inspired, according to one of its founders, by fundamental work in distributed programming accomplished by Freeman, Robson and their colleagues at Yale.) The authors also happen to have a rare ability to write about technical topics in a clear, straightforward way. Freeman and Robson are two of the best teachers around, and no one knows JavaScript better than they do. This is one of those rare books about software and computing that I can recommend without reservation."

— David Gelernter, Professor of Computer Science, Yale University

"Bullseye! Eric Freeman and Elisabeth Robson did it again. The clear, engaging, conversational tone of *Head First JavaScript Programming* coupled with accessible examples, exercises, and reader challenges, results in an indispensable tool for motivated, independent learners. Many of the illustrative problems will be familiar to readers of *Head First Learn to Code* and thus presents a unique opportunity for instructors to integrate both texts for curious and precocious students to compare and contrast languages, syntax, and fundamental concepts of each. I look forward to incorporating this new book in future student enrichment courses."

—Josh Sharfman, Shalhevet High School, Computer Science

"As a high school teacher, I highly recommend Freeman and Robson's standard-setting JavaScript guide. Their unique combination of precise computer science with engaging teaching skills makes complex concepts intuitive and accessible, turning learning into a delight for students. The innovative multi-sensory approach ensures that challenging topics are enjoyable for students of all experience levels. With Freeman and Robson, JavaScript programming becomes an adventure that blends imagination with practical skills—an essential resource for any school's computer science curriculum."

— Bryce Carlisle, co-founder of Waterloo School, Austin, Texas

"A unique combination of deep programming topics and lessons with fun applicable interactions and tasks as proof of proficiency along the way."

— Mk Haley, Themed Entertainment Association and former Disney Imagineer

Praise for other books by Eric Freeman and Elisabeth Robson

"I feel like a thousand pounds of books have just been lifted off of my head."

— Ward Cunningham, inventor of the Wiki

"This book's admirable clarity, humor and substantial doses of clever make it the sort of book that helps even non-programmers think well about problem-solving."

— Cory Doctorow, co-editor of Boing Boing, science fiction author

"The highly graphic and incremental approach precisely mimics the best way to learn this stuff..."

— Danny Goodman, author of *Dynamic HTML: The Definitive Guide* and *JavaScript & DHTML Cookbook*

Other O’Reilly books by Eric T. Freeman and Elisabeth Robson

Head First Design Patterns
Head First HTML and CSS
Head First Learn to Code
Head First HTML5 Programming

Other related books from O’Reilly

Head First HTML5 Programming
JavaScript: The Definitive Guide
JavaScript Enlightenment

Other books in O’Reilly’s *Head First* series

Head First Design Patterns
Head First HTML and CSS
Head First Java
Head First Software Architecture
Head First SQL
Head First Git
Head First C#
Head First Python
Head First jQuery
Head First Android Development
Head First Swift

Head First
JavaScript Programming

Wouldn't it be dreamy if there was a JavaScript book that was more fun than going to the dentist and more revealing than an IRS form? It's probably just a fantasy...

Eric Freeman
Elisabeth Robson

Beijing • Boston • Farnham • Sebastopol • Tokyo
O'REILLY®

Head First JavaScript Programming

by Eric Freeman and Elisabeth Robson

Copyright © 2024 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra and Bert Bates
Series Advisors:	Eric Freeman and Elisabeth Robson
Acquisitions Editor:	David Michelson
Development Editor:	Michele Cronin
Production Editor:	Christopher Faucher
Proofreader:	Rachel Head
Indexer:	Potomac Indexing, LLC
Cover Design:	Susan Thompson, based on a series design by Ellie Volckhausen
Cover and Interior Illustrations:	José Marzan Jr.

Printing History:

September 2014:	First Edition
August 2024:	Second Edition

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First JavaScript Programming*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No variables were harmed in the making of this book.

ISBN: 978-1-098-14794-5

[LSI] [2024-08-21]

To JavaScript—you weren't born with a silver spoon in your mouth, but you've outclassed every language that's challenged you in the browser.

Authors of Head First JavaScript Programming

Eric Freeman

Elisabeth Robson

Eric is described by Head First series co-creator Kathy Sierra as "one of those rare individuals fluent in the language, practice, and culture of multiple domains from hipster hacker to corporate VP, engineer, think tank."

Eric is deeply involved in education. He is currently on faculty at the University of Texas at Austin, where he teaches creative coding in the School of Design and Creative Technology. He also advises the Head First series at O'Reilly and frequently teaches live courses on their platform. This is in addition to the five books he's written for O'Reilly, covering the spectrum from beginner coding books to advanced object-oriented design.

By training, Eric is a computer scientist, having studied with industry luminary David Gelernter during his Ph.D. work at Yale University. Eric also spent a decade as a media company executive, having held the position of CTO of Disney Online and Disney.com at The Walt Disney Company.

Today, besides his academic ties, he works for WickedlySmart, an online education company.

Write to Eric at eric@wickedlysmart.com or visit his site at http://ericfreeman.com.

Elisabeth is a software engineer, writer, and trainer. She has been passionate about technology since her days as a student at Yale University, where she earned a masters of science in computer science and designed a concurrent, visual programming language and software architecture.

Elisabeth's been involved with the internet since the early days; she co-created the award-winning website, The Ada Project, one of the first websites designed to help women in computer science find career and mentorship information online.

She cofounded WickedlySmart, an online education experience centered on web technologies, where she currently creates books, articles, videos, and more. Previously, as Director of Special Projects at O'Reilly Media, Elisabeth produced in-person workshops and online courses on a variety of technical topics and developed her passion for creating learning experiences to help people understand technology. Prior to her work with O'Reilly, Elisabeth spent time spreading fairy dust at The Walt Disney Company, where she led research and development efforts in digital media.

When not in front of her computer, you'll find Elisabeth hiking, cycling, or kayaking in the great outdoors with her camera nearby, or cooking vegetarian meals.

You can send her email at beth@wickedlysmart.com.

Table of Contents (Summary)

	Intro	xxiii
1	A quick dip into JavaScript: *Getting your feet wet*	1
2	Writing real code: *Going further*	47
3	Introducing functions: *Getting functional*	83
4	Putting some order in your data: *Arrays*	131
5	Understanding objects: *A trip to Objectville*	181
6	Interacting with your web page: *Getting to know the DOM*	239
7	Types, equality, conversion, and all that jazz: *Serious types*	275
8	Bringing it all together: *Building an app*	331
9	Handling events: *Getting asynchronous*	397
10	Anonymous and higher-order functions: *Liberated functions*	445
11	Modern syntax, lexical scope, and closures: *Serious functions*	491
12	Advanced object construction: *Creating objects*	543
A	Leftovers: *The top ten things (we didn't cover)*	593
Index		609

Table of Contents (the real thing)

Intro

Your brain on JavaScript. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing JavaScript?

Who is this book for?	xxiv
We know what you're thinking	xxv
Metacognition	xxvii
Bend your brain into submission	xxix
Read me	xxx
The technical review team	xxxiii
Acknowledgments	xxxiv

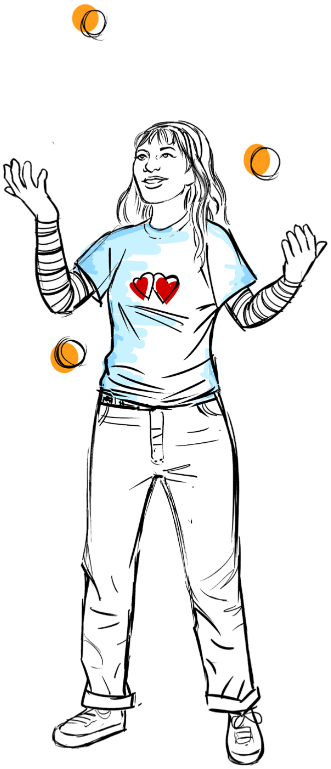
1

a quick dip into javascript

Getting your feet wet

JavaScript gives you superpowers. The **true programming language** of the web, JavaScript lets you **add behavior** to your web pages. No more dry, boring, static pages that just sit there looking at you—with JavaScript, you'll be able to reach out and touch your users, react to interesting events, grab data from the web to use in your pages, draw graphics right into those pages, and a lot more. And once you know JavaScript, you'll also be in a position to create **totally new** behaviors for your users.

You'll be in good company, too. JavaScript's not only one of the **most popular** programming languages, it's also **supported** in all modern browsers and is used in many environments outside of the browser. More on that later; for now, let's get started!



The way JavaScript works 2
How you're going to write JavaScript 3
How to get JavaScript into your page 4
JavaScript, you've come a long way... 6
How to make a statement 10
Variables and values 11
Constants, another kind of variable 12
Back away from that keyboard! 14
Express yourself 17
Doing things more than once 19
How the while loop works 20
Making decisions with JavaScript 24
And, when you need to make LOTS of decisions 25
Reach out and communicate with your user 27
A closer look at console.log 29
Opening the console 30
Coding a Serious JavaScript Application 31
How do I add code to my page? (let me count the ways!) 34
We're going to have to separate you two 35

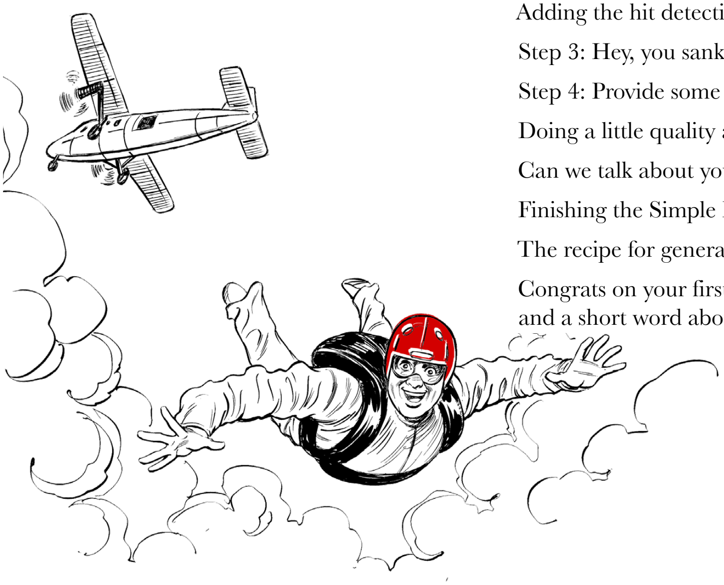
writing real code

Going further

2

You already know about variables, types, expressions... we could go on. The point is, you already know a few things about JavaScript. In fact, you know enough to write some **real code**. Some code that does something interesting, some code that someone would want to use. What you're lacking is the **real experience** of writing code, and we're going to remedy that right here and now. How? By jumping in head first and coding up a casual game, all written in JavaScript. Our goal is ambitious, but we're going to take it one step at a time. Come on, let's get this started, and if you want to launch the next startup, we won't stand in your way; the code is yours.

Let's build a Battleship game 48
First, a high-level design 49
Working through the Pseudocode 51
Oh, before we go any further, don't forget the HTML! 53
Writing the Simple Battleship code 54
Now let's write the game logic 55
Step 1: Setting up the loop, getting some input 56
How prompt works 57
Step 2: Checking the user's guess 58
Adding the hit detection code 61
Step 3: Hey, you sank my battleship! 61
Step 4: Provide some post-game analysis 62
Doing a little quality assurance 65
Can we talk about your verbosity... 69
Finishing the Simple Battleship game 70
The recipe for generating a random number 71
Congrats on your first true JavaScript program, and a short word about reusing code 75



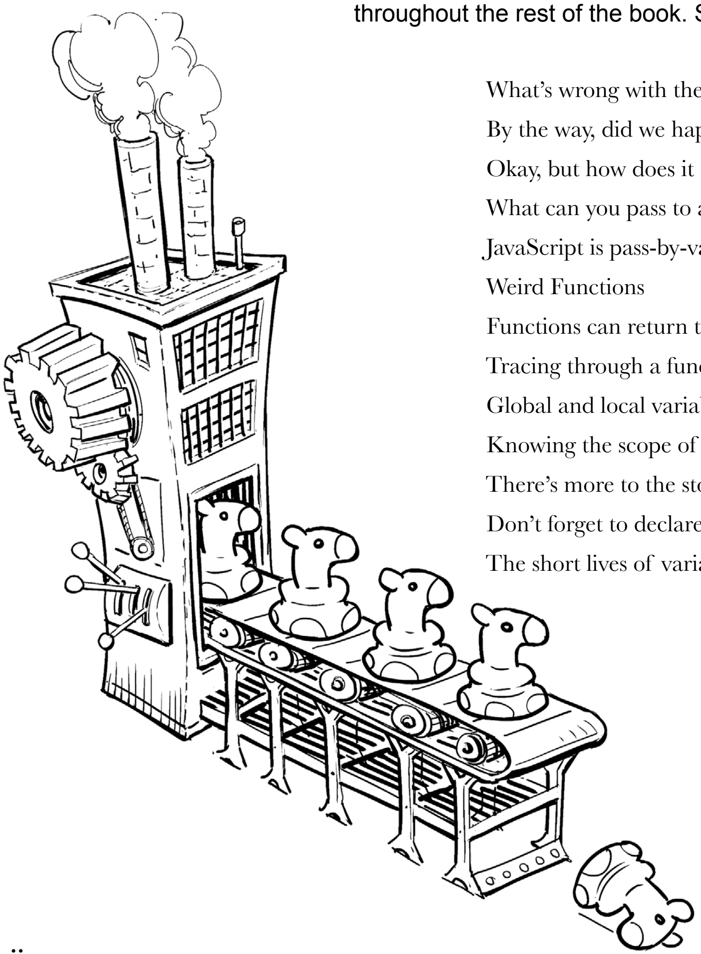
introducing functions

3

Getting functional

Get ready for your first superpower. You've got some programming under your belt; now it's time to really move things along with **functions**. Functions give you the power to write code that can be applied to all sorts of different circumstances, code that can be **reused** over and over, code that is much more **manageable,** code that can be **abstracted** away and given a simple name so you can forget all the complexity and get on with the important stuff. You're going to find not only that functions are your gateway from scripter to programmer, but that they're the key to the JavaScript programming style. In this chapter, we're going to start with the basics—the mechanics, the ins and outs of how functions really work—and then you'll keep honing your function skills throughout the rest of the book. So, let's get a good foundation started, *now*.

What's wrong with the code, anyway? 85
By the way, did we happen to mention FUNCTIONS? 87
Okay, but how does it actually work? 88
What can you pass to a function? 93
JavaScript is pass-by-value. 96
Weird Functions 98
Functions can return things too 99
Tracing through a function with a return statement 100
Global and local variables 103
Knowing the scope of your local and global variables 105
There's more to the story 106
Don't forget to declare your locals! 108
The short lives of variables 109

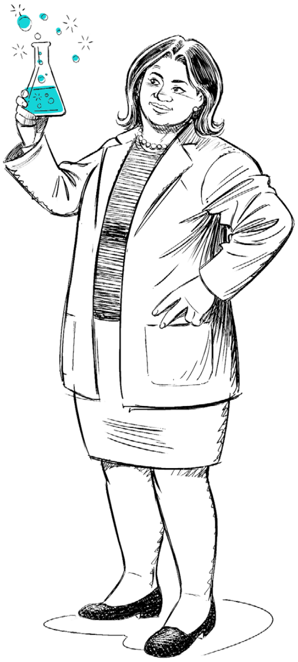


putting some order in your data

4

Arrays

There's more to JavaScript than numbers, strings, and booleans. So far you've been writing JavaScript code with **primitives**—strings, numbers, and booleans, like "Fido", 23, and true. You can do a lot with primitive types, but at some point you've got to deal with **more data**. Say, all the items in a shopping cart, or all the songs in a playlist, or a set of stars and their apparent magnitude, or an entire product catalog. For that you need a little more *oomph*. The type of choice for this kind of ordered data is a JavaScript **array**, and in this chapter we're going to walk through how to put your data into an array, how to pass it around, and how to operate on it. We'll be looking at a few other ways to **structure your data** in later chapters, but let's get started with arrays.



Can you help Bubbles-R-Us? 132
How to represent multiple values in JavaScript 132
How arrays work 134
How big is that array anyway? 136
The Phrase-O-Matic 138
Meanwhile, back at Bubbles-R-Us... 141
How to iterate over an array 144
But wait, there's a better way to iterate over an array 146
It's that time again...can we talk about your verbosity? 153
Redoing the for loop with the post-increment operator 154
Creating an array from scratch (and adding to it) 158
And the winners are... 162
A quick survey of the code 164
Writing the printAndGetHighScore function 165
Refactoring the code using printAndGetHighScore 166
Putting it all together 168

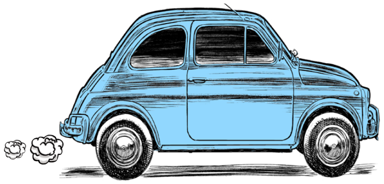
understanding objects

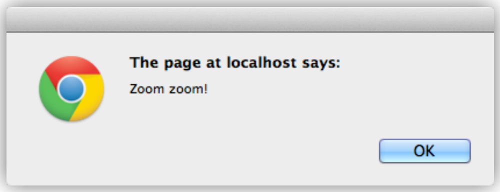
5

A trip to Objectville

So far, you've been using primitives and arrays in your code. And you've approached coding in quite a **procedural manner**, using simple statements, conditionals, and for/while loops with functions—that's not exactly **object-oriented**. In fact, it's not object-oriented *at all!* You did use a few objects here and there without really knowing it, but you haven't written any of your own objects yet. Well, the time has come to leave this boring procedural town behind and create some **objects** of your own. In this chapter, you're going to find out why using objects is going to make your life so much better—well, better in a **programming sense** (we can't really help you with your fashion sense *and* your JavaScript skills all in one book). Just a warning: once you've discovered objects, you'll never want to come back. Send us a postcard when you get there.

Did someone say "objects"?! 182
Thinking about properties... 183
How to create an object 185
What is "object-oriented" anyway? 188
How properties work 189
How does a variable hold an object? Inquiring minds want to know... 194
Comparing primitives and objects 195
Doing even more with objects 196
Does the taxi cut it? 197
Let's talk a little more about passing objects to functions 200
The Auto-O-Matic 203
Oh, behave! Or, how to add behavior to your objects 206
Improving the drive method 207
Uh-oh, not so fast... 209
Why doesn't the drive method know about the started property? 210
How "this" works 212
Method shorthand 218
How behavior affects state 219
Now let's affect the behavior with the state 220
Congrats on your first objects! 222
Guess what? There are objects all around you! 223





The page at localhost says:

Zoom zoom!

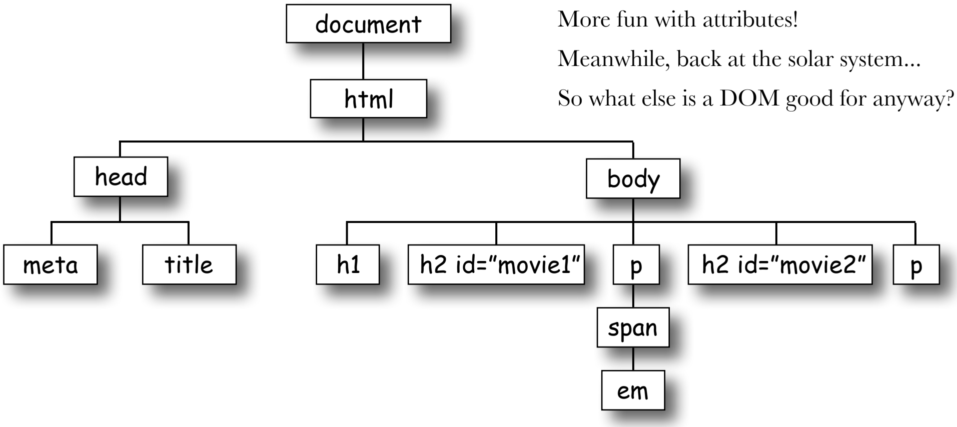
6

interacting with your web page

Getting to know the DOM

You've come a long way with JavaScript. In fact, you've evolved from a newbie to a scripter to, well, a **programmer**. But, there's something missing. To really begin leveraging your JavaScript skills, you need to know how to interact with the web page your code lives in. Only by doing that are you going to be able to write pages that are **dynamic**, pages that react, that respond, that update themselves after they've been loaded. So how do you interact with the page? By using the **DOM**, otherwise known as the **document object model**. In this chapter, we're going to break down the DOM and show you how you can use it, along with JavaScript, to teach your pages a few new tricks.

In the last chapter, we left you with a little challenge... 240
So what does the code do? 241
How JavaScript really interacts with your page 243
How to bake your very own DOM 244
A first taste of the DOM 245
Getting an element with getElementById 250
What, exactly, am I getting from the DOM? 251
Finding your inner HTML 252
What happens when you change the DOM 254
Don't even think about running my code until the page is fully loaded! 259
You say "event handler," I say "callback" 260
Why stop now? Let's take it further 264
How to set an attribute with setAttribute 265
More fun with attributes! 266
Meanwhile, back at the solar system... 267
So what else is a DOM good for anyway? 268



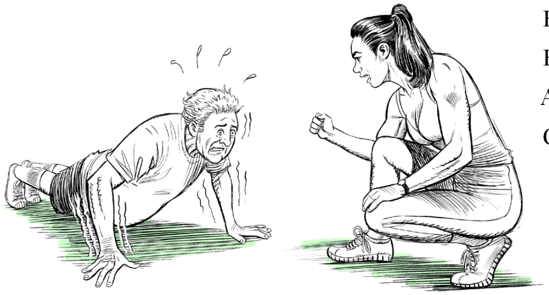
types, equality, conversion, and all that jazz

7

Serious types

It's time to get serious about our types. One of the great things about JavaScript is you can get a long way without knowing a lot of details of the language. But to truly **master the language**, get that promotion, and get on to the things you really want to do in life, you have to rock at **types**. Remember what we said about JavaScript back in Chapter 1? That it didn't have the luxury of a silver-spoon, academic, peer-reviewed language definition? Well, that's true, but the lack of an academic life didn't stop Steve Jobs and Bill Gates, and it didn't stop JavaScript either. It does mean that JavaScript doesn't have the...well, the most thought-out type system, and we'll find a few **idiosyncrasies** along the way. But don't worry, in this chapter we're going to nail all that down, and soon you'll be able to avoid all those embarrassing moments with types.

The truth is out there... 276
Watch out, you might bump into undefined... 278
How to use null 281
Dealing with NaN 283
It gets even weirder 283
We have a confession to make 285
Understanding the equality operator (otherwise known as ==) 286
How equality converts its operands 287
How to get strict with equality 290
Even more type conversions 296
How to determine if two objects are equal 299
The truthy is out there... 301
What JavaScript considers falsey 302
The Secret Life of Strings 304
How a string can look like a primitive and an object 305
How template literals work 308
A five-minute tour of string properties and methods 309
Chair Wars 314



bringing it all together

8

Building an app

Put on your toolbelt. That is, the toolbelt with all your new coding skills, your knowledge of the DOM, and even some HTML and CSS. We're going to bring everything together in this chapter to create our first true **web application**. No more **silly toy games** with one battleship and a single row of hiding places. In this chapter, we're building the **entire experience**: a nice big game board, multiple ships, and user input right in the web page. We're going to create the page structure for the game with HTML, visually style the game with CSS, and write JavaScript to code the game's behavior. Get ready: this is an all-out, pedal-to-the-metal development chapter where we're going to lay down some serious code.

This time, let's build a REAL Battleship game	332
Stepping back...to HTML and CSS	333
Creating the HTML page: the Big Picture	334
Adding some more style	338
Using the hit and miss classes	341
Designing the game	343
Implementing the view	345
The model	350
You're gonna need a bigger boat...and game board	351
How we're going to represent the ships	352
Putting it all together	358
Wait, can we talk about your verbosity again?	359
A view to a kill...	361
Implementing the controller	363
Processing the player's guess	364
Getting a player's guess	373
How to place ships	379
Avoiding a collision!	383
Congrats, it's startup time!	386

9 handling events

Getting asynchronous

After this chapter, you're going to realize you aren't in Kansas anymore. Up until now, you've been writing code that typically executes from top to bottom—sure, your code might be a little more complex than that, and make use of a few functions, objects, and methods, but at some point the code just runs its course. Now, we're awfully sorry to break this to you this late in the book, but that's **not how you typically write JavaScript code**. Rather, most JavaScript is written to **react to events**. What kind of events? Well, how about a user clicking on your page, data arriving from the network, timers expiring in the browser, changes happening in the DOM...and that's just a few examples. In fact, all kinds of events are happening **all the time**, behind the scenes, in your browser. In this chapter we're going rethink our approach to JavaScript coding, and look at how and why we should write code that reacts to events.

What are events?	399
What's an event handler?	400
Creating an event handler	401
Getting your head around events...by creating a game	404
Implementing the game	405
Let's add some more images	410
How to reuse the same handler for all the images	412
How the event object works	415
Putting the event object to work	417
Events and queues	420
How setTimeout works	424
Finishing the image game	428

10

anonymous and higher-order functions

Liberated functions

Know functions, then rock. Every art, craft, and discipline has a key principle that separates the intermediate players from the rock-star virtuosos—when it comes to JavaScript, it's truly understanding **functions** that makes the difference. Functions are fundamental to JavaScript, and many of the techniques we use to **design and organize** code depend on advanced knowledge and use of functions. The path to learning functions at this level is an interesting and often mind-bending one, so get ready...The next two chapters are going to be a bit like taking Willy Wonka's tour of the chocolate factory—you're going to encounter some wild, wacky, and wonderful things as you learn more about JavaScript functions.

The mysterious double life of the function keyword 446
How functions are values too 447
If functions are values, we can assign them to variables 450
Did we mention functions have first class status in JavaScript? 453
Taking a look at the other side of functions... 454
How to use an anonymous function 455
We need to talk about your verbosity, again 457
We can make the code even shorter with arrow functions 459
Creating arrow functions 461
Webville Cola 464
Understanding the array sort method 466
Putting it all together 467
Meanwhile, back at Webville Cola 468
Introducing higher-order functions 471
Filtering with higher-order functions 472
Don't forget your anonymous and arrow functions 473
Using reduce to get the total cases sold 476
Chaining map, filter, and reduce 478
Iterating with forEach 481

11

modern syntax, lexical scope, and closures

Serious functions

You've put functions through their paces, but there's more to learn. In this chapter, we take it further; we get hard-core. We'll show you how to up your syntax game with advanced techniques for handling arguments, parameters, and assignments. We'll then take another look at scope and some of the finer points of how JavaScript manages it. This journey through the subtleties of scope brings us to the heart of closures—a concept often shrouded in mystery but pivotal in mastering JavaScript. In the end, you'll be more expressive with your JavaScript than you thought possible.

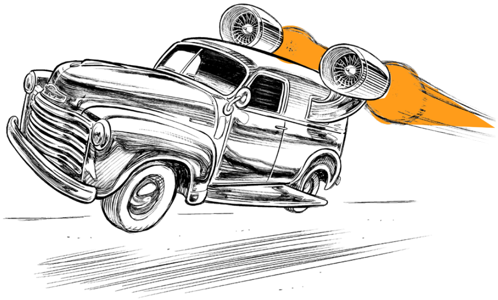
Getting serious about functional syntax 492
Spreading out your arguments 495
There's something we haven't told you about functions... 497
Function declarations are "hoisted" 498
We've done the function declarations; now we do everything else 499
We need to talk about scope 503
Taking functions beyond global scope 504
A lexical scope refresher 507
Another look at our outer/inner functions 508
Using scope for encapsulation 510
Two important JavaScript scope rules 512
Solving the mystery 516
How to make a closure 516
Using closures to implement a magic counter 521
Looking behind the curtain... 522
Implementing a counter with a closure 523
How makeTimer works 527
Implementing onlyOnceMaker 532

advanced object construction

12

Creating objects

So, far we've been crafting objects by hand. For each object, we've used an **object literal** to specify each and every property. That's okay on a small scale, but for serious code we need something better. That's where **classes** come in. With classes we can create objects much more easily, and we can create objects that all adhere to the same **design blueprint**—meaning we can use classes to ensure each object has the same properties and includes the same methods. And with classes we can write object code that is much more **concise** and a lot less error-prone when we're creating lots of objects. So, let's get started...



Creating objects with object literals 544
Using conventions for objects 545
Introducing classes 547
How to define a class 548
How to create an object from a class 549
How classes work 551
Let's add some methods 554
It's production time! 558
The basic Car class 560
Implementing the Taxi class with extends 563
Adding new methods to the Taxi class 564
Implementing the RocketCar class 567
Using an object literal to clean up our constructor 571
Reworking the Car constructor 574
Accessor properties 575
Using getters 576
What's a getter without a setter? 577
Static properties and methods 582
Counting our car production 584

A leftovers

The top ten things (we didn't cover)

We've covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small chapter. Actually, we *did* originally include everything you need to know about JavaScript programming (not already covered by the other chapters), by reducing the type point size to .00004. It all fit, but nobody could read it. So we threw most of it away, and kept the best bits for this "top ten" appendix.

#1 Modules 594
#2 JSON 596
#3 Promises 597
#4 Destructuring assignment 598
#5 Symbols and BigInt 599
#6 Map and Set 600
#7 Doing more with the DOM 603
#8 The window object 604
#9 Server-side JavaScript 605
#10 Recursion 606

Almond
MILK

how to use this book

Intro

I can't believe they put that in a JavaScript book.
Is this book for you?
If you want to learn new tools & technologies, pursue a new career or advance in your current one, and have fu
yes, it absolu
Head First
JavaScript
Programmi

In this section we answer the burning question: "So why DID they put that in a JavaScript programming book?"

Who is this book for?

If you can answer "yes" to all of these:

1. Do you have access to a computer with a modern web browser and a text editor?
2. Do you want to learn, understand, and remember how to program with JavaScript, using the best techniques and the most recent standards?
3. Do you prefer stimulating dinner party conversation to dry, dull, academic lectures?

This book is for you.

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Are you completely new to web development?
2. Are HTML and CSS foreign concepts to you? If so, you'll probably want to start with *Head First HTML and CSS* to understand how to put web pages together before tackling JavaScript.
3. Are you a kick-butt web developer looking for a *reference* book?
4. Are you afraid to try something different? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if JavaScript objects are anthropomorphized?

This book is not for you.

[Note from marketing: this book is for anyone with a credit card.]

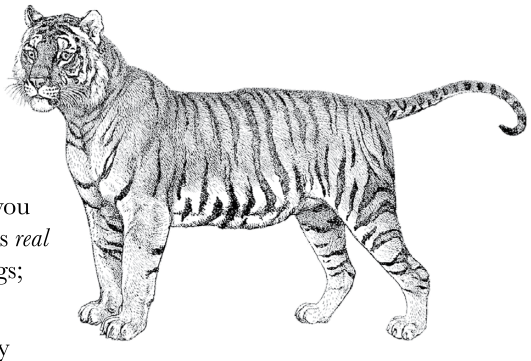
We know what you're thinking

"How can *this* be a serious JavaScript book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

We know what your *brain* is thinking



Your brain thinks THIS is important.

Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you. What happens inside your head and body?

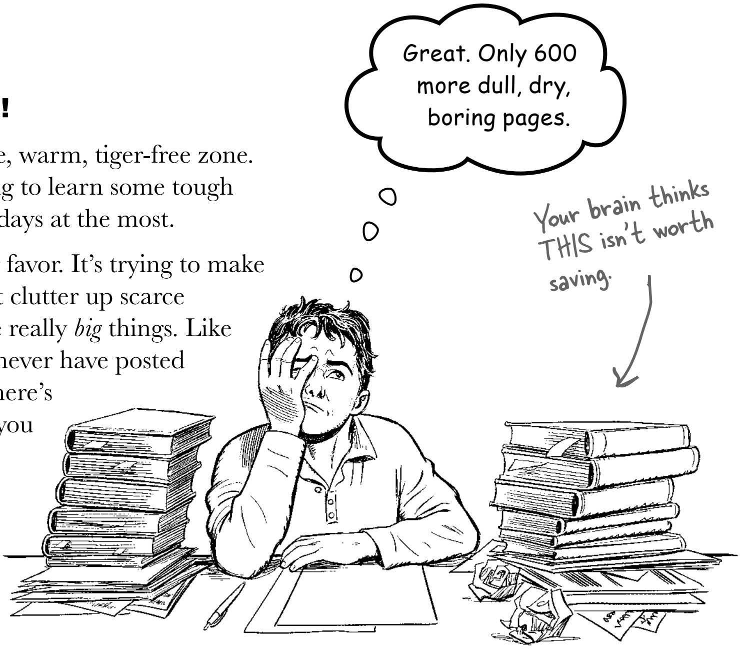
Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows...

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, 10 days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* non-important content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those "party" photos on your Facebook page. And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."


Great. Only 600 more dull, dry, boring pages.

Your brain thinks THIS isn't worth saving.

We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). They also make things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to be able to solve problems related to the content.

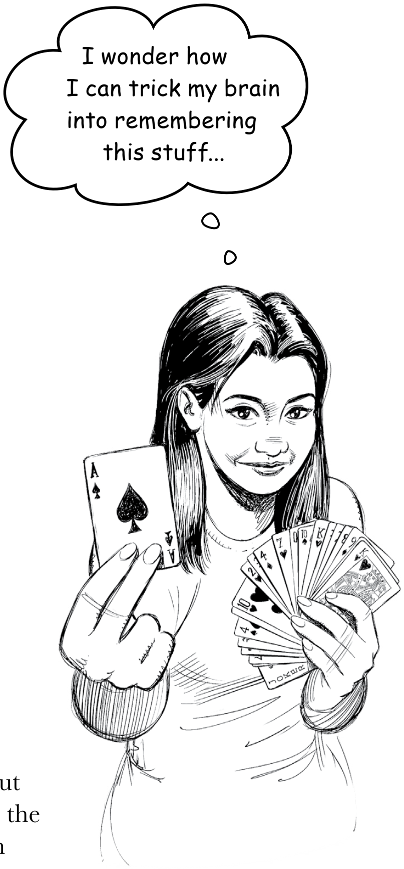
Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

Get the learner to think more deeply. Unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the...?" , and the feeling of "I rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering *doesn't*.

Metacognition: thinking about thinking


I wonder how I can trick my brain into remembering this stuff...

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to design user-friendly websites. And you probably don't want to spend a lot of time on it. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat JavaScript like it was a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important, but they keep looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else on the page, like a caption or in the body text) causes your brain to try to make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning…

Here's what WE did:

We used ***visuals***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a visual really *is* worth a thousand words. And when text and visuals work together, we embedded the text *in* the visuals because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in a paragraph somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and visuals in ***unexpected*** ways because your brain is tuned for novelty, and we used visuals and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 80 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging yet doable, because that's what most people prefer.

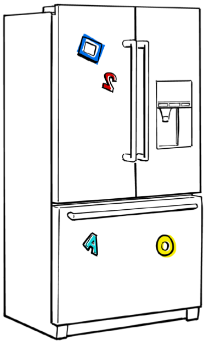
We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and we asked ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, visuals, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

Cut this out and stick it on your refrigerator.

1. **Slow down. The more you understand, the less you have to memorize.**

   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**

   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There are no Dumb Questions."**

   That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**

   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Talk about it. Out loud.**

   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

6. **Drink water. Lots of it.**

   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7. **Listen to your brain.**

   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something.**

   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Write a lot of code!**

   There's only one way to learn to program: **writing a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

Read Me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We teach modern JavaScript.

JavaScript is a programming language that didn't come up through the ivy leagues with plenty of time for academic peer review. JavaScript was thrown out into the world out of necessity and grew up in the early browser neighborhood. JavaScript has been through quite the journey to get to where it is today: a fast, brilliant, modern language that is the programming language of the web.

In this book, we explore modern JavaScript, covering the updated syntax and semantics that make coding more efficient and effective. We provide practical examples and best practices to help you master today's JavaScript and build dynamic, robust applications.

We don't exhaustively cover every single aspect of the language.

There's a lot you can learn about JavaScript. This book is not a reference book; it's a learning book, so it doesn't cover everything there is to know about JavaScript. Our goal is to teach you the fundamentals of using JavaScript so that you can pick up any old reference book and do whatever you want with JavaScript.

This book does teach you JavaScript in the browser.

The browser is not only the most common environment that JavaScript runs in, it's also the most convenient (everyone has a computer with a text editor and a browser, and that's all you need to get started with JavaScript). Running JavaScript in the browser also means you get instant gratification: you can write code and all you have to do is reload your web page to see what it does.

This book advocates well-structured and readable code based on best practices.

You want to write code that you and other people can read and understand, code that will still work in next year's browsers. You want to write code in the most straightforward way so you can get the job done and get on to better things. In this book, we're going to teach you to write clear, well-organized code that anticipates change from the get-go. Code you can be proud of, code you'll want to frame and put on the wall (just take it down before you bring your date over).

Programming is serious business. You're going to have to work, sometimes hard.

If you've already had some programming experience, then you know what we're talking about. If you're coming straight from *Head First HTML and CSS*, then you're going to find writing code is a little, well, different. Programming requires a different way of thinking. Programming is logical, at times very abstract, and requires you to think in an algorithmic way. But no worries; we're going to do all that in a brain-friendly way. Just take it a bit at a time, make sure you're well nourished, and get plenty of sleep.

That way, these new programming concepts will really sink in.

The activities are NOT optional.

The exercises and activities in this book are *not* add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. Don't skip the exercises. The crossword puzzles are the only things you don't have to do, but they're good for giving your brain a chance to think about the words in a different context.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to really get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about learning, so you'll see some of the same concepts come up more than once.

The examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of an example looking for the 2 lines they need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the examples to be robust, or even complete—they are written specifically for learning, and aren't always fully functional.

We've placed all the example files on the web so you can download them. You'll find them at http://wickedlysmart.com/hfjs.

The Brain Power exercises don't usually have answers.

For some of them, there is no right answer, and for others, part of the learning experience is for you to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

We often give you only the code, not the markup.

After we get past the first chapter or two, we often give you just the JavaScript code and assume you'll wrap it in a nice HTML wrapper. Here's a simple HTML page you can use with most of the code in this book. If we want you to use other HTML, we'll tell you:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Your HTML Page</title>
        <script>

        </script>
    </head>
    <body>

    </body>
</html>
```

← Your JavaScript code will typically go here.

← Any web page content will go here.

↑ But don't worry; at the beginning of the book we'll take you through everything.

Get the code examples, help, and discussion.

You'll find everything you need for this book online at http://wickedlysmart.com/hfjs, including code sample files.

Tech reviewers

Jeff Straw

Ismaël Martin "Bing" Demiddel

These guys really rocked it; they were there for us throughout the review process and provided invaluable, detailed feedback on everything!

Frank D. Moore

Alfred J. Speller

Bruce Forkush

Javier Ruedas

Thank you to our amazing review team

This book has been more carefully reviewed than any of our previous books. Over 270 people participated in reading and critiquing the first edition of this book in real time as we wrote it. This worked better than we ever imagined and was instrumental in shaping every aspect of *Head First JavaScript Programming*. Our heartfelt thanks to everyone who participated; the book is so much better because of you.

For the first edition, the amazing technical reviewers pictured above provided feedback above and beyond, and each made significant contributions to the book. The following reviewers also made contributions across different aspects of the book: **Galina N. Orlova, J. Patrick Kelley, Claus-Peter Kahl, Rob Cleary, Rebeca Dunn-Krahn, Olaf Schoenrich, Jim Cupec, Matthew M. Hanrahan, Russell Alleen-Willems, Christine J. Wilson, Louis-Philippe Breton, Timo Glaser, Charmaine Gray, Lee Beckham, Michael Murphy, Dave Young, Don Smallidge, Alan Rusyak, Eric R. Liscinsky, Brent Fazekas, Sue Starr, Eric (Orange Pants) Johnson, Jesse Palmer, Manabu Kawakami,** Alan McIvor, Alex Kelley, Yvonne Bichsel Truhon, Austin Throop, Tim Williams, J. Albert Bowden II, Rod Shelton, Nancy DeHaven Hall, Sue McGee, Francisco Debs, Miriam Berkland, Christine H Grecco, Elhadji Barry, Athanasios Valsamakis, Peter Casey, Dustin Wollam, and Robb Kerley.

For the second edition, we had a fabulous group of technical reviewers who provided valuable feedback as we updated this book to include new features added to the language since the first edition. Thank you to Adam Scott, Maya Shavin, Stoyan Stefanov, and Edward Wong for your feedback.

Acknowledgments for the first edition

We're also extremely grateful to our esteemed technical reviewer, **David Powers**. The truth is we don't write books without David anymore, he's just saved our butts too many times. It's getting a little like Elton and Bernie; we're starting to ask ourselves if we actually could write a book without him. David ~~helps us~~ forces us to make the book more sound and technically accurate, and his second career as a standup comic really comes in handy when we're tuning the more humorous parts of the book. Thank you once again, David—you're the ultimate professional, and we sleep better at night knowing we've passed your technical muster.

Esteemed Reviewer, David Powers

Don't let the smile fool you, this guy is hard-core (technically, of course).

Meghan Blanchette

A huge, massive thanks to our editor for the first edition, **Meghan Blanchette**, who cleared the path for this book, removed every obstacle to its completion, waited patiently, and sacrificed family time to get it done. She helped keep us sane during the book writing process, which is probably the best thing an editor can do for an author.

And another big shoutout to our Chief Editor Emeritus, **Mike Hendrickson**, who spearheaded this book from the very beginning. Thanks again, Mike; none of our books would have happened without you. You've been our champion for well over a decade and we love you for it!

Mike Hendrickson

Acknowledgments for the second edition

Our editor:

Michele Cronin

Our editor for the second edition, **Michele Cronin**, helped us through every stage of updating this book and put up with our questions, delays, and asks with the utmost grace. Once again, we were lucky to have an editor who helped keep us sane while being patient throughout the process.

Thank you so much, Michele!

The O'Reilly team:

It takes a whole team of people to write a book, and we couldn't do it without everyone. Thank you especially to:

Mary Tresseler, Melissa Duffield, David Michelson, Kristen Brown, Chris Faucher, Rachel Head, and Ellie Volkhausen.

The large number of acknowledgments is because we're testing the theory that everyone mentioned in a book acknowledgment will buy at least one copy, probably more, what with relatives and everything. If you'd like to be in the acknowledgments of our next book, and you have a large family, write to us.

1 a quick dip into javascript

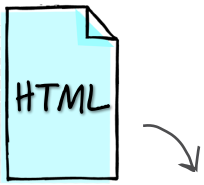
Getting your feet wet

JavaScript gives you superpowers. The **true programming language** of the web, JavaScript lets you **add behavior** to your web pages. No more dry, boring, static pages that just sit there looking at you—with JavaScript, you'll be able to reach out and touch your users, react to interesting events, grab data from the web to use in your pages, draw graphics right into those pages, and a lot more. And once you know JavaScript, you'll also be in a position to create **totally new** behaviors for your users.

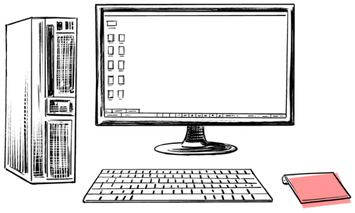
You'll be in good company, too. JavaScript's not only one of the **most popular** programming languages, it's also **supported** in all modern browsers and is used in many environments outside of the browser. More on that later; for now, let's get started!

The way JavaScript works

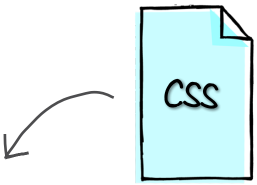
If you're used to creating structure, content, layout, and style in your web pages, isn't it time you added a little behavior as well? After all, there's no need for the page to just *sit there*. Great pages should be *interactive* and *dynamic*. That's where JavaScript comes in. Let's start by taking a look at how JavaScript fits into the *web page ecosystem*:



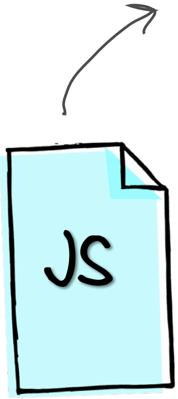
You use HTML, or Hypertext Markup Language, to specify all the **content** of your pages along with their **structure**, like paragraphs, headings, and sections.



Browser



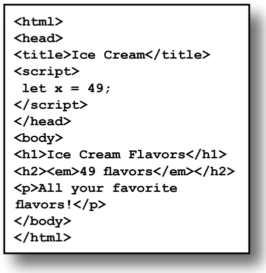
And you use CSS, or Cascading Style Sheets, to specify how the HTML is presented...the colors, fonts, borders, margins, and layout of your page. CSS gives you **style**, and it does it in a way that is separate from the structure of the page.



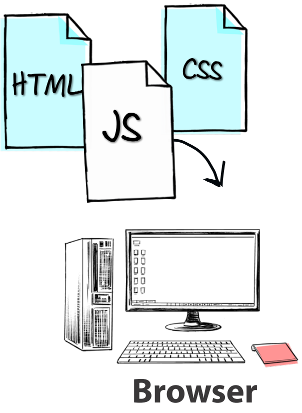
JavaScript—HTML and CSS's computational cousin—lets you create **behavior** in your web pages. Need to react when a user clicks on your "On Sale for the next 30 seconds!" button? Double-check your user's form input on the fly? Grab some data from NASA and display it? Or how about play a game? Look to JavaScript. JavaScript gives you a way to add code to your page so that you can compute, react, draw, communicate, alert, alter, update, change, and we could go on...anything dynamic, that's JavaScript in action.

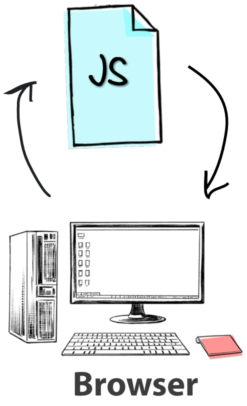
How you're going to write JavaScript

JavaScript is *the* programming language for the web browser. With your typical programming language, you have to write the code, compile it, link it, and deploy it. With JavaScript, all you need to do is write code right into your page, and then load it into a browser. From there, the browser will happily begin executing your code. Let's take a closer look at how this works:



```
<html>
<head>
<title>Ice Cream</title>
<script>
 let x = 49;
</script>
</head>
<body>
<h1>Ice Cream Flavors</h1>
<h2><em>49 flavors</em></h2>
<p>All your favorite
flavors!</p>
</body>
</html>
```





❶ Writing

You create your page just like you always do, with HTML content and CSS styles. And you also include JavaScript in your page. As you'll see, just like with HTML and CSS, you can put everything together in one file, or you can place the JavaScript in its own file, to be included in your page.

We'll talk about the best way in a bit.

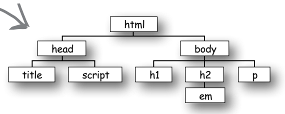
❷ Loading

Point your browser to your page, just as you always do. The browser sees the code and begins parsing it immediately, getting ready to execute it. Note that like with HTML and CSS, if the browser sees errors in your code, it will do its best to keep moving and reading more JavaScript, HTML, and CSS. The last thing it wants to do is not be able to give the user a page to see.

❸ Executing

The browser starts executing your code as soon as it encounters the code in your page, and continues executing it for the lifetime of your page. Some code executes once and is not executed again (until you reload the page); other code is executed whenever the user does something, like click a button or move the mouse. You'll see examples of both in this book.

For future reference, the browser also builds an "object model" of your HTML page that JavaScript can make use of. Put that in the back of your brain, we'll come back to it later...



How to get JavaScript into your page

First things first. You can't get very far with JavaScript if you don't know how to get it into a web page. So, how do you do that? Using the `<script>` element, of course!

Let's take a boring old garden-variety web page and add some dynamic behavior using a `<script>` element. At this point, don't worry too much about the details of what we're putting into the `<script>` element—your goal right now is to get some JavaScript working.

Here's our standard HTML doctype, and <html> and <head> elements.

And we've got a pretty generic <body> for this page as well.

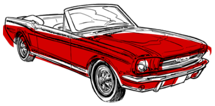
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Just a Generic Page</title>
    <script>
      setTimeout(wakeUpUser, 5000);
      function wakeUpUser() {
         alert("Are you going to stare at this boring page forever?");
      }
    </script>
  </head>
  <body>
      <h1>Just a generic heading</h1>
      <p>Not a lot to read about here. I'm just an obligatory paragraph living in
an example in a JavaScript book. I'm looking for something to make my life more
exciting.</p>
  </body>
</html>
```

Ah, but we've added a <script> element to the <head> of the page.

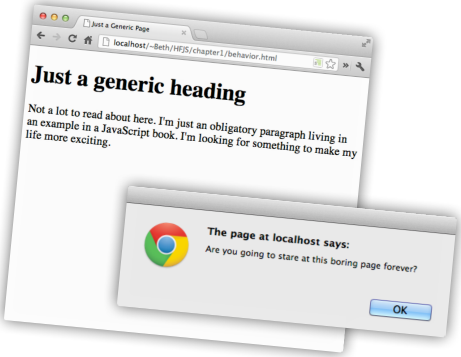
And we've written some JavaScript code inside it.

Again, don't worry too much about what this code does. Then again, we bet you'll want to take a look at the code and see if you can think through what each part might do.

A little test drive



Go ahead and type this page into a file named "behavior.html". Drag the file to your browser (or use File > Open) to load it. What does it do? Hint: you'll need to wait five seconds to find out.

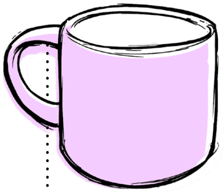


Just a generic heading

Not a lot to read about here. I'm just an obligatory paragraph living in an example in a JavaScript book. I'm looking for something to make my life more exciting.

The page at localhost says:

Are you going to stare at this boring page forever?



Relax

Just relax. At this point, we don't expect you to read JavaScript like you grew up with it. In fact, all we want you to do right now is get a feel for what JavaScript looks like.

That said, you're not totally off the hook, because we need to get your brain revved up and working. Remember that code on the previous page? Let's just walk through it to get a feel for what it might do:

Perhaps a way to count five seconds of time? Hint: 1,000 milliseconds = 1 second.

A way to create reusable code and call it "wakeUpUser"?

```
setTimeout(wakeUpUser, 5000);
function wakeUpUser() {
   alert("Are you going to stare at this boring page forever?");
}
```

Clearly a way to alert the user with a message.

there are no Dumb Questions

Q: Is JavaScript related to Java?

A: Only by name. JavaScript was created during a time when Java was a red-hot popular language, and the inventors of JavaScript capitalized on that popularity by making use of the Java name. Both languages borrow some syntax from programming languages like C, but other than that, they are quite different.

Q: I've heard JavaScript is a bit of an underpowered language. Is it?

A: JavaScript certainly wasn't a powerlifter in its early days, but its importance to the web has grown since then, and as a result, many resources (including brain power from some of the best minds in the business) have gone into supercharging the performance of JavaScript. But, you know what? Even before JavaScript was super fast, it was always a brilliant language. As you'll see, we're going to do some very powerful things with it.

Q: Is JavaScript the best way to create dynamic web pages?

A: JavaScript is **the** language for programming in the browser. There are variants of JavaScript, like TypeScript, but the TypeScript code you write will be translated into JavaScript before it runs in the browser. With today's super-fast JavaScript environments and sophisticated APIs, JavaScript is here to stay and is the standard for programming in the browser.

Q: My friend is using JavaScript inside a music application... or at least he says he is. Is that possible?

A: Yes! JavaScript has broken out of the browser as a general scripting language for many applications, from graphics utilities to music applications and even to server-side programming. Your investment in learning JavaScript is likely to pay off in ways beyond web pages.

Q: You say that many other languages are compiled. What exactly does that mean, and why isn't JavaScript?

A: With conventional programming languages like C, C++, or Java, you compile the code before you execute it. Compiling takes your code and produces a machine-efficient representation of it, usually optimized for runtime performance. Scripting languages are typically *interpreted*, which means that the browser runs each line of JavaScript code as it gets to it. Scripting languages place less importance on runtime performance and are more geared toward tasks like prototyping, interactive coding, and flexibility. This was the case with early JavaScript, and it was why, for many years, the performance of JavaScript was not so great. There is a middle ground, however; an interpreted language can be compiled on the fly, and that's the path browser manufacturers have taken with modern JavaScript. In fact, with JavaScript you now have the conveniences of a scripting language, while enjoying the performance of a compiled language. By the way, we'll use the words *interpret*, *evaluate,* and *execute* in this book. They have slightly different meanings in various contexts, but for our purposes, they all basically mean the same thing.

JavaScript, you've come a long way...

JavaScript 1.0

Netscape might have been before your time, but it was the first *real* browser company. Back in the mid-1990s browser competition was fierce, particularly with Microsoft, and so adding new, exciting features to the browser was a priority.

Toward that goal, Netscape wanted to create a scripting language that would allow anyone to add scripts to their pages. Enter LiveScript, a language developed in short order to meet that need. Now, if you've never heard of LiveScript, that's because this was all about the time that Sun Microsystems introduced Java, and as a result drove its stock to stratospheric levels. So, why not capitalize on that success and rename LiveScript to JavaScript? After all, who cares if they don't actually have anything to do with each other? Right?

Did we mention Microsoft? It created its own scripting language soon after Netscape did, named, um, JScript, and it was, um, quite similar to JavaScript. And so began the browser wars and a frustrating time for developers.

JavaScript 2015

After all that confusion, JavaScript finally grew up. ECMAScript, an official language definition for JavaScript was born, and now serves as the standard language definition for all JavaScript implementations (in and out of the browser).

By 2015, JavaScript had finally come of age and gained the respect of professional developers. Having a solid standard helped, along with serious commitment from browser makers like Google, which pushed JavaScript into the professional limelight with Google Maps and other complex browser-based applications.

With all the new attention, many of the best programming language minds focused on improving JavaScript's interpreters and made improvements to its runtime performance. After JavaScript 2015, a major language update, we switched from official version numbers for language releases to yearly updates.

JavaScript 2024

Today, JavaScript is *the* language of the web. Pre-compilers make interpreting JavaScript code in webpages blindingly fast. Syntax checkers, syntax-aware code editors, and robust browser-based debugging tools have professionalized web development. JavaScript is one of the most popular languages in the world and has been implemented in environments as diverse as embedded systems, web servers, and music-making applications.

The language is fairly mature at this point. Most language updates are small and incorporated quickly into browsers. Numerous JavaScript libraries and frameworks mean that you can do almost anything with JavaScript that you can do in any other language.

After a strange and contentious history, JavaScript has made it!

1995
2015
2024

Look how easy it is to write JavaScript

You don't know JavaScript yet, but we bet you can make some good guesses about how JavaScript code works. Take a look at each line of code and see if you can guess what it does. Write in your answers below. We've done one for you to get you started. If you get stuck, the answers are on the next page.

```
let price = 28.99;
let discount = 10;
let total =
      price - (price * (discount / 100));
if (total > 25) {
   freeShipping();
}

let count = 10;
while (count > 0) {
   juggle();
   count = count - 1;
}

const dog = {name: "Rover", weight: 35};
if (dog.weight > 30) {
    alert("WOOF WOOF");
} else {
    alert("woof woof");
}

let circleRadius = 20;
let circleArea =
   Math.PI * (circleRadius * circleRadius);
```

Create a variable named price, and assign the value 28.99 to it.

Solution on page 8

From page 7

Look how easy it is to write JavaScript Solution

You don't know JavaScript yet, but we bet you can make some good guesses about how JavaScript code works. Take a look at each line of code and see if you can guess what it does. Here are our answers.

```
let price = 28.99;

let discount = 10;

let total =
      price - (price * (discount / 100));

if (total > 25) {

   freeShipping();

}
```

Create a variable named price, and assign the value 28.99 to it.

Create a variable named discount, and assign the value 10 to it.

Compute a new price by applying a discount and then assign it to the variable total.

Compare the value in the variable total to 25. If it's greater...

...then do something with freeShipping.

End the if statement.

```
let count = 10;

while (count > 0) {

   juggle();

   count = count - 1;

}
```

Create a variable named count, and assign the value 10 to it.

As long as the variable count is greater than 0...

...do some juggling, and...

...reduce the value of count by 1 each time.

End the while loop.

```
const dog = {name: "Rover", weight: 35};

if (dog.weight > 30) {

    alert("WOOF WOOF");

} else {

    alert("woof woof");

}
```

Create a constant named dog with a name and weight.

If the dog's weight is greater than 30...

...alert "WOOF WOOF" to the browser's web page.

Otherwise...

...alert "woof woof" to the browser's web page.

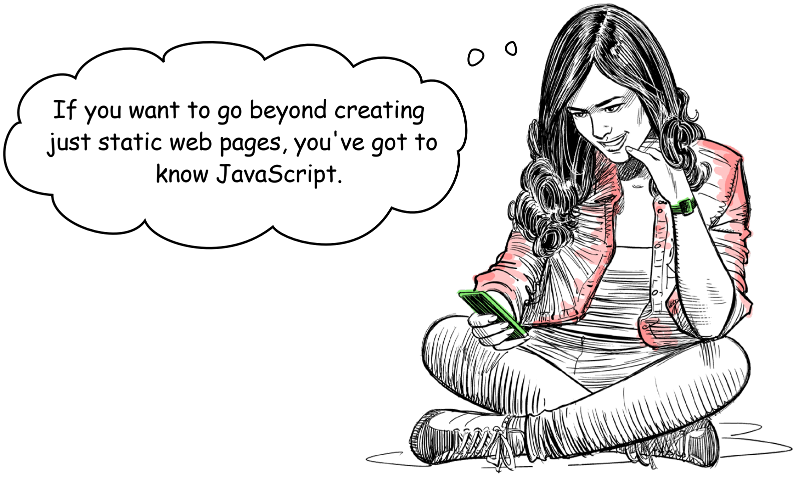
End the if/else statement.

```
let circleRadius = 20;

let circleArea =

   Math.PI * (circleRadius * circleRadius);
```

Create a variable, circleRadius, and assign the value 20 to it.

Create a variable named circleArea...

...and assign the result of this expression to it (1256.6370614359173).


If you want to go beyond creating just static web pages, you've got to know JavaScript.

With HTML and CSS, you can create some great-looking pages. But once you know JavaScript, you can really expand on the kinds of pages you can create.

Knowing JavaScript might increase the size of your paycheck too!

So much so, in fact, you might actually start thinking of your pages as applications (or even experiences!) rather than mere pages.

Now, you might be saying, “I already know that, why do you think I’m reading this book?” Well, we actually wanted to use this opportunity to have a little chat about learning JavaScript. If you already have a programming language or scripting language under your belt, then you have some idea of what lies ahead. However, if you’ve mostly been using HTML and CSS to date, you should know that there is something fundamentally different about learning a programming language.

With HTML and CSS, what you’re doing is largely declarative—for instance, you’re declaring, say, that some text is a paragraph or that all the HTML elements in the “sale” class should be colored red. With JavaScript, you’re adding *behavior* to the page, and to do that you need to describe computation. You’ll need to be able to describe things like “calculate the user’s score by summing up all the correct answers” or “do this action 10 times” or “when the user clicks on that button play the you-have-won sound” or even “go off and get the current temperature, and put it in this page.”

To do those things, you need a language that is quite different from HTML or CSS. Let’s see how…

How to make a statement

When you create HTML, you usually **mark up** text to give it structure. To do that, you add elements, attributes, and values to the text:

```
<h1 class="drink">Mocha Caffe Latte</h1>
<p>Espresso, steamed milk, and chocolate syrup,
just the way you like it.</p>
```

We're using HTML to mark up text to create structures like headings and paragraphs. Like, "I need a large heading called Mocha Caffe Latte; it's a heading for a drink. And I need a paragraph after that."

CSS is a bit different. With CSS, you're writing a set of **rules**, where each rule selects elements in the page and then specifies a set of styles for those elements:

```
h1.drink {
    color: brown;
}
p {
    font-family: sans-serif;
}
```

With CSS we write rules that use selectors, like h1.drink and p, to determine what parts of the HTML the style is applied to.

Let's make sure all drink headings are colored brown...

...and we want all the paragraphs to have a sans-serif type font.

With JavaScript, you write **statements**. Each statement specifies a small part of a computation, and together, all the statements create the behavior of a page:

A set of statements.

Each statement does a little bit of work, like declaring some variables to contain values for us.

```
let age = 25;
let name = "Owen";

if (age > 14) {
    alert("Sorry this page is for kids only!");
} else {
    alert("Welcome " + name + "!");
}
```

Here we create a variable to contain an age of 25, and we also need a variable to contain—or, as we usually say, store—the value "Owen".

Or making decisions, such as: Is the age of the user greater than 14?

And if so alerting the user they are too old for this page.

Otherwise, we welcome the user by name, like this: "Welcome Owen!" (But since Owen is 25, we don't do that in this case.)

Variables and values

You might have noticed that JavaScript statements usually involve variables. Variables are used to store values. What kinds of values? Here are a few examples:

```
let winners = 2;
```

This statement declares a variable named winners and assigns a numeric value of 2 to it.

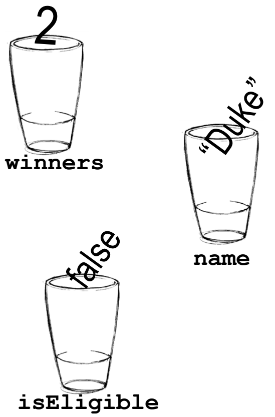
```
let name = "Duke";
```

This one assigns a string of characters to the variable name (we call those "strings," for short).

```
let isEligible = false;
```

And this statement assigns the value false to the variable isEligible. We call true/false values "booleans."

Pronounced "boo-lee-ans."



There are other values that variables can hold beyond numbers, strings, and booleans, and we'll get to those soon enough—but no matter what a variable contains, we create all variables the same way. Let's take a closer look at how to declare a variable:

We start with the let keyword when declaring a variable whose value can change.

Even if JavaScript doesn't complain when you leave off the let, you should declare your variables. We'll tell you why later...

Next, we give the variable a name.

```
let winners = 2;
```

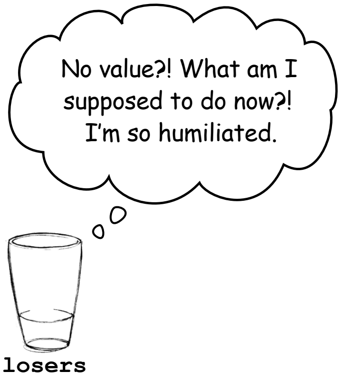
We always end an assignment statement with a semicolon.

And, optionally, we assign a value to the variable by adding an equals sign followed by the value.

We say optionally, because if you want, you can create a variable without an initial value, and then assign it a value later. To create a variable without an initial value, just leave off the assignment part, like this:

```
let losers;
```

By leaving off the equals sign and value, you're just declaring the variable for later use.


No value?! What am I supposed to do now?! I'm so humiliated.
losers

Brain Power

If you declare a variable without a value, such as:

We're using let to declare a new variable.

We're calling the variable flavor.

```
let flavor;
```

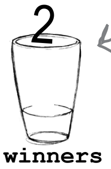
But we're not giving it an initial value.

what value do you think JavaScript assigns to this variable?

Constants, another kind of variable

So far, we've used the keyword **`let`** to declare our variables. And that's typically what we want to do with variables whose values can *vary,* or in other words, change their value over time. For instance, if we use `let` to declare the variable **`winners`** and assign it the value 2, we can change the value in `winners` later to be 3 if another winner comes along:

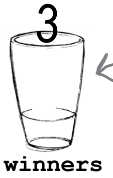
```
let winners = 2;
```



We initially set the value of winners to 2 when we declare the variable.

Later on, more people play the game and another person wins. So, we change the value of winners.

```
winners = 3;
```



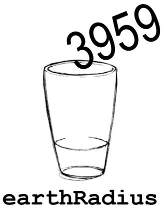
Now the value of winners has changed to 3.

Notice we don't redeclare winners here, we only assign a new value. If you try to redeclare winners, you'll get an error.

Sometimes, however, we do not want the values in our variables to vary at all. There are situations in which we might want to give a name to a value that we'll use in our code, but we don't ever want that value to change. Here's a good example: the radius of planet Earth. It might be handy to assign this value to a variable so we can use `earthRadius` instead of the number in our code. We don't want anyone to come along and change this value accidentally, so how can we make sure the value of `earthRadius` never changes? We can use a *constant* instead of a variable, like this:

```
const earthRadius = 3959;
```

This statement declares a constant with the keyword const named earthRadius and assigns a numeric value of 3959 to it (we're measuring the radius of the Earth in miles here).

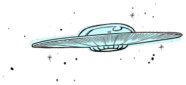


Note: the value in a constant doesn't vary, so we call them constants, not variables.

If you try to assign a new value to earthRadius you'll get an error.

```
earthRadius = 9000;
```

This will not work!



No evil space aliens can come along and change the radius of Earth in our code. Whew!

Sharpen your pencil

Identify the declarations below that you think are best suited for let and for const. We've done a couple for you. Check our solution at the end of the chapter before you go on.

const `DISTANCE_TO_MOON = 238900;`

let `last_guess = 0;`

______ `SCREEN_WIDTH = 1024;`

______ `counter = 100;`

______ `firstUSPresident = "Washington";`

______ `fluxCapacitorReading = "System Normal";`

Solution on page 41

Back away from that keyboard!

You know variables have a name, and you know they have a value.

You also know some of the things a variable can hold are numbers, strings, and boolean values.

But what can you call your variables? Is any name okay? Well, no, but the rules around creating variable names are simple—just follow the two rules below to create valid variable names:

1. **Start your variable names with a letter or an underscore.**
2. **After that, use as many letters, numeric digits, underscores, or dollar signs as you like.**

Oh, and one more thing; we really don't want to confuse JavaScript by using any of the built-in *keywords*, like **let** or **function** or **false**, so consider those off-limits for your own variable names. We'll get to some of these keywords and what they mean later in the book, but here's a list to take a quick look at:

break
case
catch
class
const
continue
debugger
default
delete
do
else
enum
export
extends
false
finally
for
function
if
implements
import
in
instanceof
interface
let
new
package
private
protected
public
return
static
super
switch
this
throw
true
try
typeof
var
void
while
with
yield

there are no Dumb Questions

Q: What's a keyword?

A: A keyword is a reserved word in JavaScript. JavaScript uses these reserved words for its own purposes, and it would be confusing to you and the browser if you started using them for your variables and constants.

Q: Why aren't we using var to declare our variables?

A: Earlier versions of JavaScript used a single keyword, var, to declare all variables. Now, the var keyword is no longer recommended and has been largely replaced by let and const. These keywords work slightly differently to var and have some benefits over var that we'll come back to later.

Q: What if I used a keyword as part of my variable name? For instance, can I have a variable named ifOnly (that is, a variable that contains the keyword if)?

A: You sure can; just don't match the keyword exactly. It's also good to write clear code, so in general you wouldn't want to use something like `elze`, which might be confused with `else`.

Q: Is JavaScript case sensitive? In other words, are myvariable and MyVariable the same thing?

A: If you're used to HTML markup, you might be used to case-insensitive languages; after all, <head> and <HEAD> are treated the same by the browser. With JavaScript, however, case matters for variable, constant, keyword, and function names, and pretty much everything else, too. So pay attention to your use of upper- and lowercase.

WEBVILLE TIMES

How to avoid those embarassing naming mistakes

You've got a lot of flexibility in choosing your variable and constant names, so here are a few Webville tips to make naming easier:

Choose names that mean something.

Variable names like _m, $, r, and foo might mean something to you, but they are generally frowned upon in Webville. Not only are you likely to forget them over time, but your code will be much more readable with names like angle, currentPressure, and passedExam.

Use "camel case" when creating multiword variable names.

At some point you're going to have to decide how you name a variable that represents, say, a two-headed dragon with fire. Just use camel case, in which you capitalize the first letter of each word (other than the first): twoHeadedDragonWithFire. Camel case is easy to form, widely spoken in Webville, and gives you enough flexibility to create as specific a variable name as you need. There are other schemes too, but this is one of the more commonly used (even beyond JavaScript). For constants, use an underscore between words.

Use variable names that begin with _ and $ only with very good reason.

Variables that begin with $ are usually reserved for JavaScript libraries, and while some authors use variables beginning with _ for various conventions, we recommend you stay away from both unless you have very good reason (you'll know if you do).

Be safe.

Be safe in your variable naming. We'll cover a few more tips for staying safe later in the book, but for now, be clear in your naming, avoid keywords, and always use let when declaring a variable and const when declaring a constant.

Syntax Fun

- Each statement ends in a semicolon.

```
x = x + 1;
```

- A single-line comment begins with two forward slashes. Comments are just notes to you or other developers about the code.They aren't executed.

```
// I'm a comment
```

- Whitespace doesn't matter (almost everywhere).

```
x     =     2233;
```

- Surround strings of characters with double quotes (or single; both work, just be consistent).

```
"You rule!"
'And so do you!'
```

- Don't use quotes around the boolean values true and false.

```
rockin = true;
```

- Variables don't have to be given a value when they are declared.

```
let width;
```

- JavaScript, unlike HTML markup, is case sensitive, meaning upper- and lowercase matters. The variable `counter` is different from the variable `Counter`.

BE the Browser

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. After you've done the exercise, look at the solution at the end of the chapter to see if you found them all.

A

```
// Test for jokes
const joke = "JavaScript walked into a bar....';
let toldJoke = "false";
let $punchline =
  "Better watch out for those semi-colons."
let %entage = 20;
let result

if (toldJoke == true) {
    Alert($punchline);
} else
    alert(joke);
}
joke = "Knock, knock. Who's there? JavaScript....";
```

Don't worry too much about what this JavaScript does for now; just focus on looking for errors in variables and syntax.

B

```
\\ Movie Night
let zip code = 98104;
const joe'sFavoriteMovie = Forbidden Planet;
let movieTicket$    =    9;

if (movieTicket$ >= 9) {
    alert("Too much!");
} else {
    alert("We're going to see " + joe'sFavoriteMovie);
}
```

→ Solution on page 42

Express yourself

To truly express yourself in JavaScript, you need ***expressions***. Expressions evaluate to values. You've already seen a few in passing in our code examples. Take the expression in this statement, for instance:

This expression evaluates to a price reduced by a discount that is a percentage of the price. So, if the price is 10 and the discount is 20, we get 8 as a result.

Here's a JavaScript statement that assigns the result of evaluating an expression to the variable total.

We use * for multiply and / for divide.

```
let total = price - (price * (discount / 100));
```

Here's our variable, total.

And the assignment.

And this whole thing is an expression.

If you've ever taken a math class, balanced your checkbook, or done your taxes, we're sure these kinds of numeric expressions are nothing new.

There are also string expressions; here are a few:

```
"Dear " + "Reader" + ","
```

This adds together, or concatenates, these strings to form a new string, "Dear Reader,".

```
"super" + "cali" + youKnowTheRest
```

Same here, except we have a variable that contains a string as part of the expression. This evaluates to "supercalifragilisticexpialidocious".

```
phoneNumber.substring(0,3)
```

Just another example of an expression that results in a string. We'll get to exactly how this works later, but this returns the area code of a US phone number string.

We also have expressions that evaluate to **true** or **false**, otherwise known as boolean expressions. Work through each of these to see how you get true or false from them:

```
age < 14
```

If a person's age is less than 14, this is true; otherwise, it's false. We could use this to test if someone is a child or not.

```
cost >= 3.99
```

If the cost is 3.99 or greater, this is true. Otherwise, it's false. Get ready to buy on sale when it's false!

```
animal == "bear"
```

This is true when animal contains the string "bear". If it does, beware!

Expressions can evaluate to a few other types, too; we'll get to these later in the book. For now, the important thing is to realize all these expressions evaluate to something: a value that is a number, a string, or a boolean. Let's keep moving and see what that gets you!

Sharpen your pencil

Get out your pencil and put some expressions through their paces. For each expression below, compute its value and write in your answer. Yes, WRITE IN…forget what your Mom told you about writing in books and scribble your answer right in this book! Be sure to check your answers at the end of the chapter.

Can you say “Celsius to Fahrenheit calculator”?

`(9 / 5) * temp + 32`

What is the result when temp is 10? __________

This is a boolean expression. The == operator tests if two values are equal to each other.

`color == "orange"`

Is this expression true or false when color has the value “pink”? __________
Or has the value “orange”? __________

`name + ", " + "you've won!"`

What value does this compute to when name is “Martha”?
____

This tests if the first value is greater than the second. You can also use >= to test if the first value is greater than or equal to the second.

`yourLevel > 5`

When yourLevel is 2, what does this evaluate to? ________
When yourLevel is 5, what does this evaluate to? ________
When yourLevel is 7, what does this evaluate to? ________

`(level * points) + bonus`

Okay, level is 5, points is 30,000, and bonus is 3,300. What does this evaluate to? ________

`color != "orange"`

Is this expression true or false when color has the value “pink”? __________

The != operator tests if two values are NOT equal to each other.

Extra CREDIT!

`1000 + "108"`

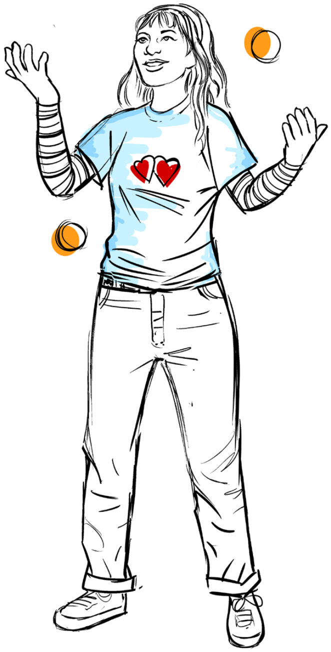
Are there a few possible answers? Only one is correct. Which would you choose? ______________________

Solution on page 43

Serious Coding

Did you notice that the = operator is used in assignments, while the == operator tests for equality? That is, we use one equals sign to assign values to variables. We use two equals signs to test if two values are equal to each other. Substituting one for the other is a common coding mistake.

```
while (juggling) {
    keepBallsInAir();
}
```



Doing things more than once

You do a lot of things more than once:

Lather, rinse, repeat…

Wax on, wax off…

Eat candies from the bowl until they're all gone.

Of course, you'll often need to do things in code more than once, and JavaScript gives you a few ways to repeatedly execute code in a loop: **while**, **for**, **for in**, **for of**, and **forEach**. Eventually, we'll look at all these ways of looping, but let's focus on **while** for now.

We just talked about expressions that evaluate to boolean values, like `scoops > 0`, and these kinds of expressions are the key to the while statement. Here's how:

A while statement starts with the keyword while.

while uses a boolean expression that we call a conditional test, or conditional for short.

If the conditional is true, everything in the code block is executed.

```
while (scoops > 0) {
    console.log("Another scoop!");
    scoops = scoops - 1;
}
```

What's a code block? Everything between the curly braces; that is, between { }.

And, if our conditional is true, then after we execute the code block, we loop back around and do it all again. If the conditional is false, we're done.

Like we said, lather, rinse, repeat!

How the while loop works

Seeing as this is your first while loop, let's trace through a round of its execution to see exactly how it works. Notice we've added a declaration for scoops to declare it and initialize it to the value 5.

Now let's start executing this code. First, we set scoops to five:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

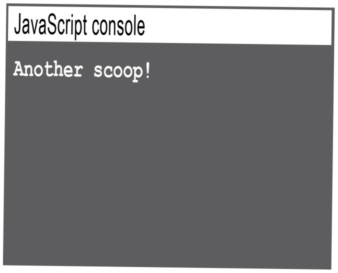
After that, we hit the while statement. When we evaluate a while statement, the first thing we do is evaluate the conditional to see if it's true or false:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

Is scoops greater than zero? Looks like it to us!

Because the conditional is true, we start executing the block of code. The first statement in the body writes the string "Another scoop!" to the browser's console:*

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```



JavaScript console

```
Another scoop!
```

* We'll show you how to access the console shortly.

The next statement subtracts one from the number of scoops and then sets scoops to that new value, four:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

1 scoop gone, 4 left!

That's the last statement in the block, so we loop back up to the conditional and start over again:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

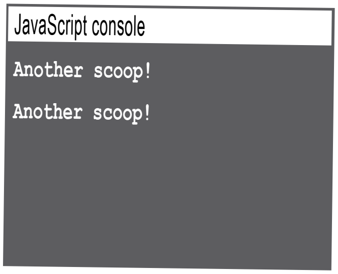
Evaluating our conditional again, this time scoops is four. But that's still more than zero:

Still plenty left!

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

Once again, we write the string "Another scoop!" to the browser:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

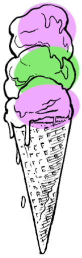


JavaScript console

```
Another scoop!
Another scoop!
```

The next statement subtracts one from the number of scoops and sets scoops to that new value, which is three:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```



2 scoops gone,
3 left!

That's the last statement in the block, so we loop back up to the conditional and start over again:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

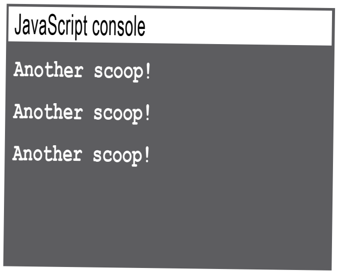
Evaluating our conditional again, this time scoops is three. But that's still more than zero:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

Still plenty left!

Once again, we write the string "Another scoop!" to the browser:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```



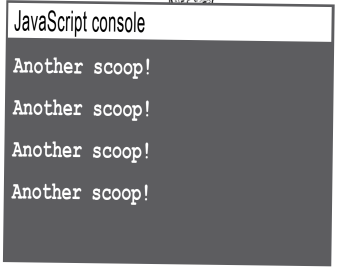
JavaScript console

```
Another scoop!
Another scoop!
Another scoop!
```

And as you can see, this continues...each time we loop, we decrement (reduce scoops by 1), write another string to the browser, and keep going:

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!<br>");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

3 scoops gone, 2 left!

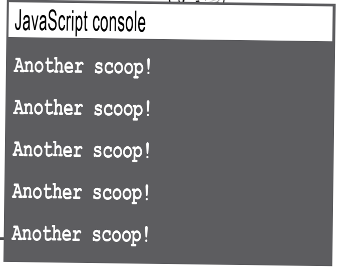


```
JavaScript console
Another scoop!
Another scoop!
Another scoop!
Another scoop!
```

And continues...

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

4 scoops gone, 1 left!

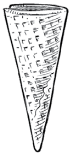


```
JavaScript console
Another scoop!
Another scoop!
Another scoop!
Another scoop!
Another scoop!
```

Until the last time...this time, something's different. Scoops is zero, so our conditional returns false. That's it, folks; we're not going to go through the loop anymore, we're not going to execute the block. This time, we bypass the block and execute the statement that follows it:

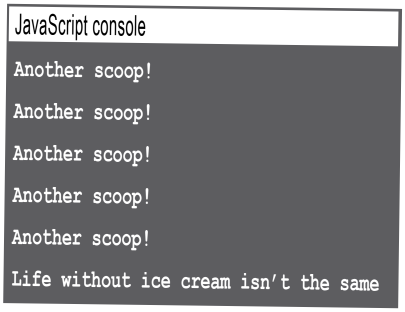
```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!<br>");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```

5 scoops gone, 0 left!

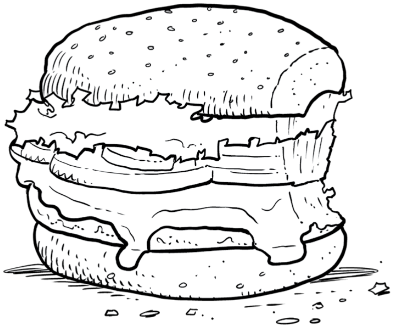


Now we execute the other console.log and write the string "Life without ice cream isn't the same". We're done!

```
let scoops = 5;
while (scoops > 0) {
   console.log("Another scoop!<br>");
   scoops = scoops - 1;
}
console.log("Life without ice cream isn't the same");
```



```
JavaScript console
Another scoop!
Another scoop!
Another scoop!
Another scoop!
Another scoop!
Life without ice cream isn't the same
```



```
if (cashInWallet > 5)  {
   order = "I'll take a cheeseburger and fries";
} else {
   order = "I'll just have a glass of water";
}
```

How do you make decisions like "Do I have enough cash for dinner?" in code?

Making decisions with JavaScript

You've just seen how you use a conditional to decide whether to continue looping in a `while` statement. You can also use boolean expressions to make decisions in JavaScript with the `if` statement. The `if` statement executes the code inside its curly braces (its *code block*) only if a conditional test is true. Here's an example:

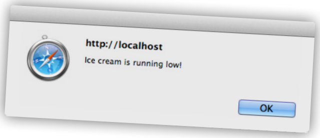
Here's the if keyword, followed by a conditional and a block of code.

This conditional tests to see if we're down to fewer than three scoops.

```
if (scoops < 3) {
   alert("Ice cream is running low!");
}
```

And if we've got fewer than three left, then we execute the if statement's code block.

alert takes a string and displays it in a pop-up dialog in your browser. Give it a try!


http://localhost

Ice cream is running low!

With an `if` statement, we can also string together multiple tests by adding on one or more `else ifs`, like this:

```
if (scoops >= 5) {
   alert("Eat faster, the ice cream is going to melt!");
} else if (scoops < 3) {
   alert("Ice cream is running low!");
}
```

We can have one test, and then another test with if/else if.

Add as many tests with "else if" as you need, each with its own associated code block that will be executed when the condition is true.

And, when you need to make LOTS of decisions...

You can string together as many `if/else` statements as you need, and if you want one, even a final catch-all `else`, so that if all conditions fail, you can handle it. Like this:

In this code, we check to see if there are five or more scoops left...

```
if (scoops >= 5) {
   alert("Eat faster, the ice cream is going to melt!");
} else if (scoops == 3) {
   alert("Ice cream is running low!");
} else if (scoops == 2) {
   alert("Going once!");
} else if (scoops == 1) {
   alert("Going twice!");
} else if (scoops == 0) {
   alert("Gone!");
} else {
   alert("Still lots of ice cream left, come and get it.");
}
```

...or if there are precisely three left...

...or if there are 2, 1, or 0, and then we provide the appropriate alert.

And if none of the conditions above are true, then this code is executed.

there are no Dumb Questions

Q: What exactly is a code block?

A: Syntactically, a code block (which we usually just call a block) is a set of statements, which could be one statement, or as many as you like, grouped together between curly braces. Once you've got a block of code, all the statements in that block are treated as a group to be executed together in sequence. For instance, all the statements within the block in a while statement are executed if the condition of the while is true. The same holds for a block in an if or else if.

Q: I've seen code where the conditional is a variable with a value that isn't a boolean, like a string or a number. How does that work?

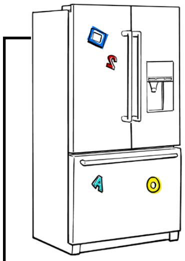
A: We'll be covering that a little later, but the short answer is JavaScript is quite flexible in what it thinks is a true or false value. For instance, any variable that holds a (nonempty) string is considered true, but a variable that hasn't been set to a value is considered false. We'll get into these details soon enough.

Q: You've said that expressions can result in things other than numbers, strings, and booleans. Like what?

A: Right now we're concentrating on what are known as the *primitive types*; that is, numbers, strings, and booleans. Later we'll take a look at more complex types, like arrays, which are collections of values, objects, and functions.

Q: Where does the name boolean come from?

A: Booleans are named after George Boole, an English mathematician who invented Boolean logic. You'll often see boolean written "Boolean," to signify that these types of variables are named after George.



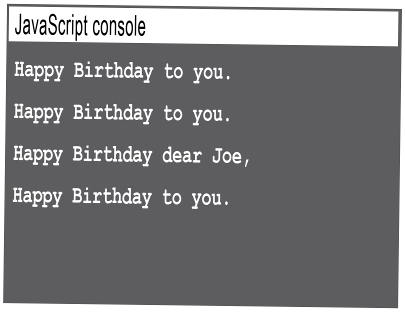
Code Magnets

A JavaScript program is all scrambled up on the fridge. Can you put the magnets back in the right places to make a working program to produce the output shown below? Check your answer at the end of the chapter before you go on.

Arrange these magnets to make a working JavaScript program.

```
console.log("Happy Birthday dear " + name);
console.log("Happy Birthday to you.");
let i = 0;
const name = "Joe";
i = i + 1;
}
console.log("Happy Birthday to you.");
while (i < 2) {
```

Your unscrambled program should produce this output in the console.



JavaScript console

```
Happy Birthday to you.
Happy Birthday to you.
Happy Birthday dear Joe,
Happy Birthday to you.
```

Solution on page 44

Reach out and communicate with your user

We've been talking about making your pages more interactive, and to do that you need to be able to communicate with your user. As it turns out, there are a few ways to do that, and you've already seen a couple of them. Let's get a quick overview and then we'll dive into these in more detail throughout the book:

Create an alert.

As you've seen, the browser gives you a quick way to alert your users through the `alert` function. Just call `alert` with a string containing your alert message, and the browser will give your user the message in a nice dialog box. A small confession, though: we've been overusing this because it's easy; `alert` really should be used only when you truly want to stop everything and let the user know something.

We're using these two methods in this chapter.

Write directly into your document.

Think of your web page as a document (that's what the browser calls it). You can use a function called `document.write` to write arbitrary HTML and content into your page at any point. In general, this is considered bad form, although you'll see it used here and there in old code.

Use the console.

Every JavaScript environment also has a console that can log messages from your code. We've been using this in our examples so far. To write a message to the console's log, you use the function `console.log` and hand it a string that you'd like printed to the log (more details on using the console log in just a moment). You can view `console.log` as a great tool to get started with JavaScript and for troubleshooting your code, but typically your users will never see your console log, so it's not a very effective way to communicate with them.

The console is a really handy way to help find errors in your code! If you've made a typing mistake, like missing a quote, JavaScript will usually give you an error in the console to help you track it down.

Directly manipulate your document.

This is the big leagues; this is the way you want to be interacting with your page and users. Using JavaScript, you can access your actual web page, read and change its content, and even alter its structure and style! This all happens by making use of your browser's *document object model*. As you'll see, this is the best way to communicate with your users. Using the document object model requires knowledge of how your page is structured and of the programming interface that is used to read and write to the page. We'll be getting there soon enough—but first, we've got some more JavaScript to learn.

This is what we're working toward. When you get there, you'll be able to read, alter, and manipulate your pages in any number of ways.

Who Does What?

All our methods of communication have come to the party with masks on. Can you help us unmask each one? Match the descriptions on the right to the names on the left. We've done one for you.

document.write

console.log

alert

document object model

I'll stop your user in her tracks and deliver a short message. The user has to click "OK" to go further.

I can insert a little HTML and text into a document. I'm not the most elegant way to get a message to your users, and I'm being phased out of browsers.

Using me, you can totally control a web page: get values that a user typed in, alter the HTML or the style, or update the content of your page.

I'm just here for simple debugging purposes. Use me, and I can write out information to a special developer's console.

→ Solution on page 45

A closer look at console.log

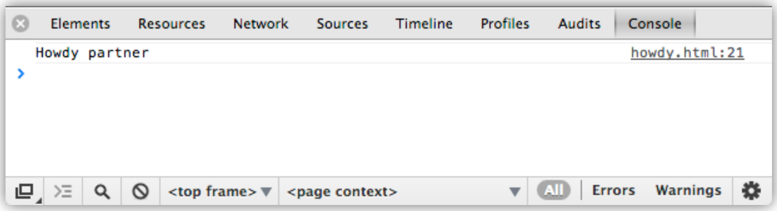
Let's take a closer look at how `console.log` works so we can use it in this chapter to see the output from our code, and throughout the book to inspect the output of our code and debug it. Remember, though, the console is not a browser feature most casual users of the web will encounter, so you won't want to use it in the final version of your web page. Writing to the console log is typically done to troubleshoot as you develop your page. That said, it's a great way to see what your code is doing while you're learning the basics of JavaScript. Here's how it works:

Take any old string...

```
let message = "Howdy" + " " + "partner";
console.log(message);
```

...and give it to console.log, and it will be shown in the browser's console, like this.

The console contains all the output logged by your code.



```
Howdy partner                                             howdy.html:21
>
```

there are no Dumb Questions

Q: I get that console.log can be used to output strings, but what exactly is it? I mean, why are the "console" and the "log" separated by a period?

A: Ah, good point. We're jumping ahead a bit, but think of the console as an object that does things—console-like things. One of those things is logging, and to tell the console to log for us, we use the syntax "console.log" and pass it our output in between parentheses. Keep that in the back of your mind; we'll come back to talk a lot more about objects a little later in the book. For now, you've got enough to use console.log.

Q: Can the console do anything other than log?

A: Yes, but typically people just use it to log. There are a few more advanced ways to use log (and console), but they tend to be browser-specific. Note that a console is something all modern browsers supply, but it isn't part of any formal specification.

Q: Uh, the console looks great, but where do I find it? I'm using it in my code and I don't see any output!

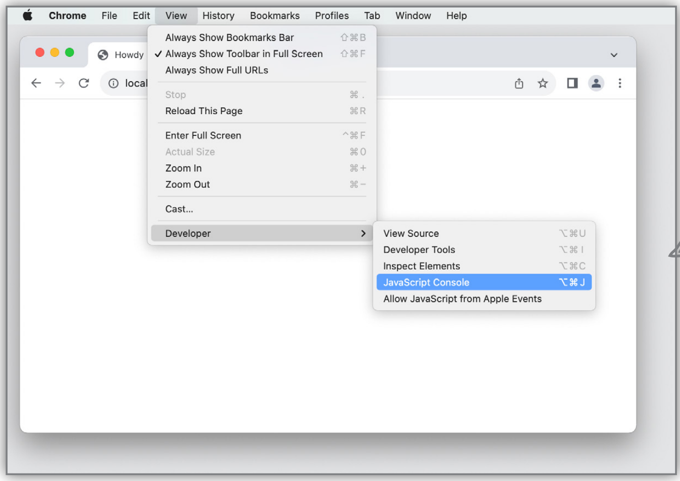
A: In most browsers, you have to explicitly open the console window. Check out the next page for details.

Opening the console

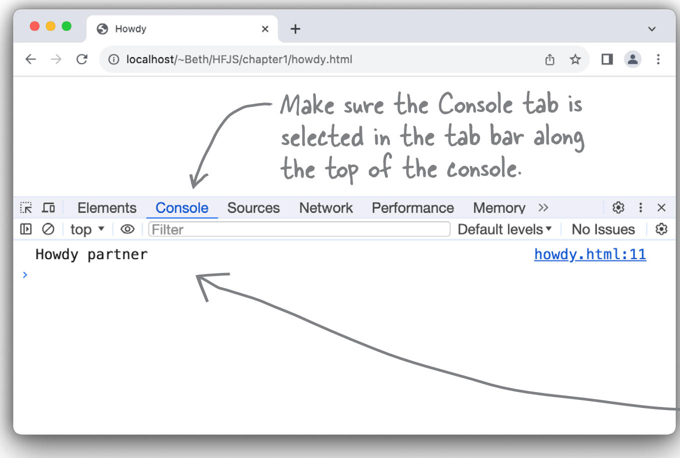
Every browser has a slightly different implementation of the console. And to make things even more complicated, the way that browsers implement the console changes fairly frequently—not in a huge way, but enough so that by the time you read this, your browser's console might look a bit different from what we're showing here.

We're going to show you how to access the console in the Chrome browser (version 123) on macOS, and we'll put instructions on how to access the console in all the major browsers online at https://wickedlysmart.com/hfjsconsole. Once you get the hang of the console in one browser, it's fairly easy to figure out how to use it in other browsers too, and we encourage you to try using the console in at least two browsers so you're familiar with them.

Note: you don't need to type the Howdy code in. We're just showing where the console is. We'll start typing in code in just a sec...


Chrome
File
Edit
View
History
Bookmarks
Profiles
Tab
Window
Help
Howdy
local
Always Show Bookmarks Bar ⇧⌘B
✓ Always Show Toolbar in Full Screen ⇧⌘F
Always Show Full URLs
Stop ⌘.
Reload This Page ⌘R
Enter Full Screen ^⌘F
Actual Size ⌘0
Zoom In ⌘+
Zoom Out ⌘−
Cast...
Developer
View Source ⌥⌘U
Developer Tools ⌥⌘I
Inspect Elements ⌥⌘C
JavaScript Console ⌥⌘J
Allow JavaScript from Apple Events

To access the console in Chrome (on a Mac), use the View > Developer > JavaScript Console menu item.


Howdy
localhost/~Beth/HFJS/chapter1/howdy.html
Elements
Console
Sources
Network
Performance
Memory
top
Filter
Default levels
No Issues
howdy.html:11

```
Howdy partner
```

Make sure the Console tab is selected in the tab bar along the top of the console.

The console will appear in the bottom part of your browser window.

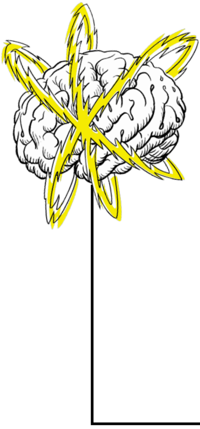
Don't worry about what these other tabs are for. They're useful, but the most important one for now is Console, so we can see console.log messages from our code.

You should see any messages you give to console.log in your code displayed in the window here.

Coding a Serious JavaScript Application

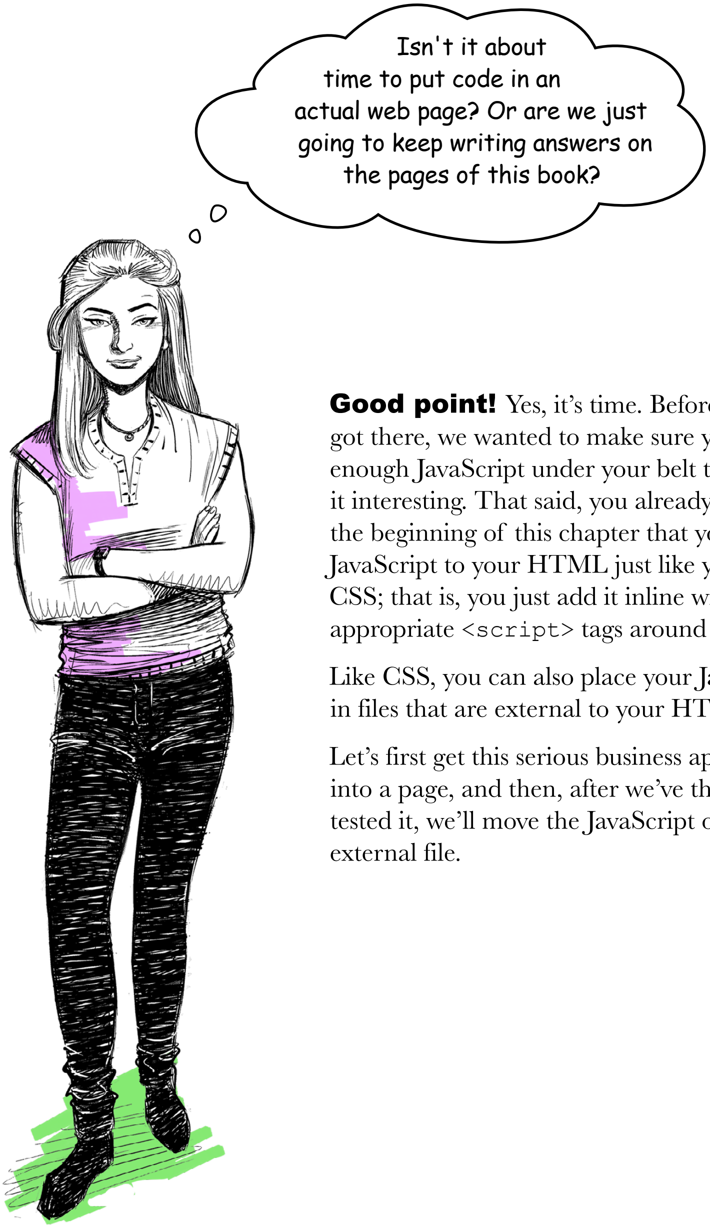
Let's put all these new JavaScript skills and `console.log` to good use with something practical. We need some variables, a `while` statement, and some `if` statements with `else`s. Add a little more polish, and we'll have a super-serious business application before you know it. But before you look at the code, think to yourself how you'd code that classic favorite, "99 bottles of rootbeer."

```
const word = "bottles";
let count = 99;
while (count > 0) {
    console.log(count + " " + word + " of rootbeer on the wall");
    console.log(count + " " + word + " of rootbeer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
        console.log(count + " " + word + " of rootbeer on the wall.");
    } else {
        console.log("No more " + word + " of rootbeer on the wall.");
    }
}
```



Brain Power

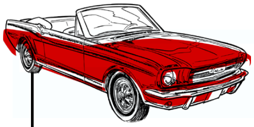
There's a little flaw in our code. It runs correctly, but the output isn't 100% perfect. See if you can find the flaw, and fix it.


Isn't it about time to put code in an actual web page? Or are we just going to keep writing answers on the pages of this book?

Good point! Yes, it's time. Before we got there, we wanted to make sure you had enough JavaScript under your belt to make it interesting. That said, you already saw at the beginning of this chapter that you add JavaScript to your HTML just like you add CSS; that is, you just add it inline with the appropriate `<script>` tags around it.

Like CSS, you can also place your JavaScript in files that are external to your HTML.

Let's first get this serious business application into a page, and then, after we've thoroughly tested it, we'll move the JavaScript out to an external file.

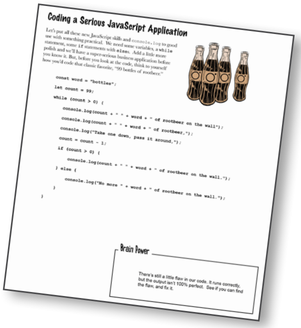


Test Drive

Okay, let's get some code in the browser...follow the instructions below to get your serious business app launched! You'll see our result below.

To download all the code and sample files for this book, please visit https://wickedlysmart.com/hfjs.

① Check out the HTML below; that's where your JavaScript's going to go. Go ahead and type in the HTML, and then place the JavaScript from two pages back between the <script> tags. You can use an editor like Notepad (Windows) or TextEdit (Mac), making sure you are in plain text mode. Or, if you have a favorite HTML editor, like VS Code or WebStorm, you can use that too.



Coding a Serious JavaScript Application

Brain Power

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>

    </script>
  </body>
</html>
```

Type this in.

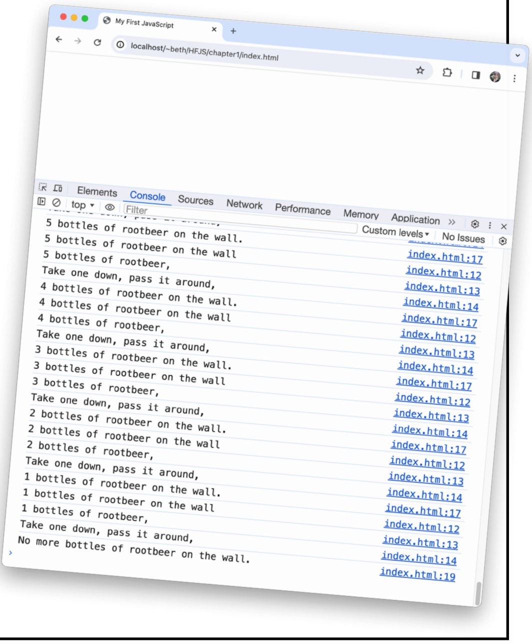
Here are the <script> tags. At this point you know that's where you should put your code.

② Save the file as "index.html".

③ Load the file into your browser. You can either drag the file right on top of your browser window, or use the File > Open (or File > Open File) menu option in your favorite browser.

④ You won't see anything in the web page itself because we're logging all the output to the console, using console.log. So, open up the browser's console, and congratulate yourself on your serious business application.

Here's our test run of this code. The code creates the entire lyrics for the 99 bottles of rootbeer song and logs the text to the browser's console.



```
5 bottles of rootbeer on the wall.          index.html:17
5 bottles of rootbeer on the wall           index.html:12
5 bottles of rootbeer,                      index.html:13
Take one down, pass it around,              index.html:14
4 bottles of rootbeer on the wall.          index.html:17
4 bottles of rootbeer on the wall           index.html:12
4 bottles of rootbeer,                      index.html:13
Take one down, pass it around,              index.html:14
3 bottles of rootbeer on the wall.          index.html:17
3 bottles of rootbeer on the wall           index.html:12
3 bottles of rootbeer,                      index.html:13
Take one down, pass it around,              index.html:14
2 bottles of rootbeer on the wall.          index.html:17
2 bottles of rootbeer on the wall           index.html:12
2 bottles of rootbeer,                      index.html:13
Take one down, pass it around,              index.html:14
1 bottles of rootbeer on the wall.          index.html:17
1 bottles of rootbeer on the wall           index.html:12
1 bottles of rootbeer,                      index.html:13
Take one down, pass it around,              index.html:14
No more bottles of rootbeer on the wall.    index.html:19
```

How do I add code to my page? (let me count the ways!)

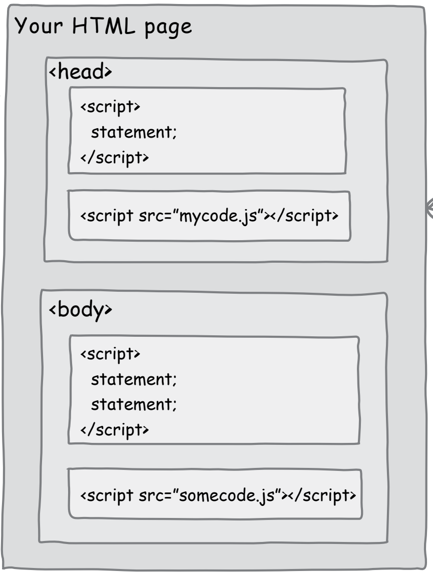
You already know you can add the `<script>` element with your JavaScript code to the `<head>` or `<body>` of your page, but there are a couple of other ways to add your code to a page. Let's check out all the places you can put JavaScript (and why you might want to put it one place over another):

You can place your code inline, in the <head> element. The most common way to add code to your pages is to put a <script> element in the <head>. Sure, it makes your code easy to find and seems to be a logical place for your code, but it's not always the best place. Why? Read on...

Or, you can add your code inline in the body of the document. To do this, enclose your JavaScript code in the <script> element and place it in the <body> of your page (typically at the end of the body).

This is a little better. Why? When your browser loads a page, it loads everything in your page's <head> before it loads the <body>. So, if your code is in the <head>, users might have to wait a while to see the page. If the code is loaded after the HTML in the <body>, users will get to see the page content while they wait for the code to load.

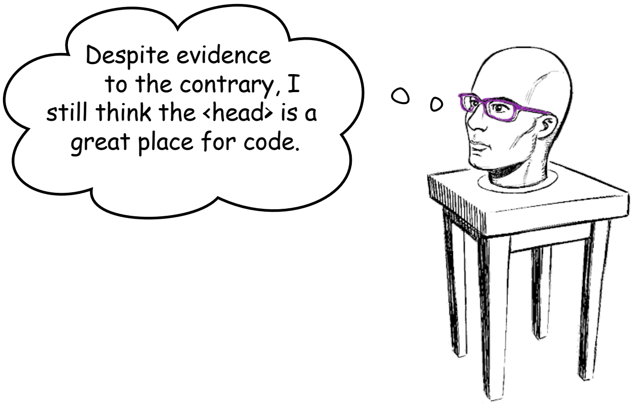
Still, is there a better way? Read on...



Or, put your code in its own file and link to it from the <head>. This is just like linking to a CSS file. The only difference is that you use the src attribute of the <script> tag to specify the URL to your JavaScript file.

When your code is in an external file, it's easier to maintain (separately from the HTML) and can be used across multiple pages. But this method still has the drawback that all the code needs to be loaded before the body of the page. Is there a better way? Read on...

Finally, you can link to an external file in the body of your page. Ahhh, the best of both worlds. You have a nice, maintainable JavaScript file that can be included in any page, and it's referenced from the bottom of the body of the page, so it's only loaded after the body of the page. Not bad.


Despite evidence to the contrary, I still think the <head> is a great place for code.

We're going to have to separate you two

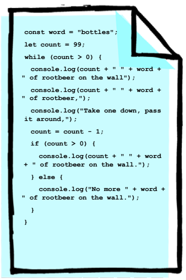
Going separate ways hurts, but we know we have to do it. It's time to take your JavaScript and move it into its own file. Here's how you do that…

① Open "index.html" and select all the code; that is, everything between the <script> tags. Your selection should look like this:

Select just the code, not the <script> tags; you won't need those where you're going…

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>
      const word = "bottles";
      let count = 99;
      while (count > 0) {
         console.log(count + " " + word + " of rootbeer on the wall");
         console.log(count + " " + word + " of rootbeer,");
         console.log("Take one down, pass it around,");
         count = count - 1;
         if (count > 0) {
            console.log(count + " " + word + " of rootbeer on the wall.");
         } else {
            console.log("No more " + word + " of rootbeer on the wall.");
         }
      }
    </script>
  </body>
</html>
```

② Now create a new file named "code.js" in your editor, and place the code into it. Then save "code.js" in the same folder as your HTML.



```
const word = "bottles";
let count = 99;
while (count > 0) {
  console.log(count + " " + word +
" of rootbeer on the wall");
  console.log(count + " " + word +
" of rootbeer,");
  console.log("Take one down, pass
it around,");
  count = count - 1;
  if (count > 0) {
    console.log(count + " " + word
+ " of rootbeer on the wall.");
  } else {
    console.log("No more " + word +
" of rootbeer on the wall.");
  }
}
```

"code.js"

③ You need to place a reference to the "code.js" file in "index.html" so that it's retrieved and loaded when the page loads. To do that, delete the JavaScript code from "index.html", but leave the <script> tags. Then add a src attribute to your opening <script> tag to reference "code.js".

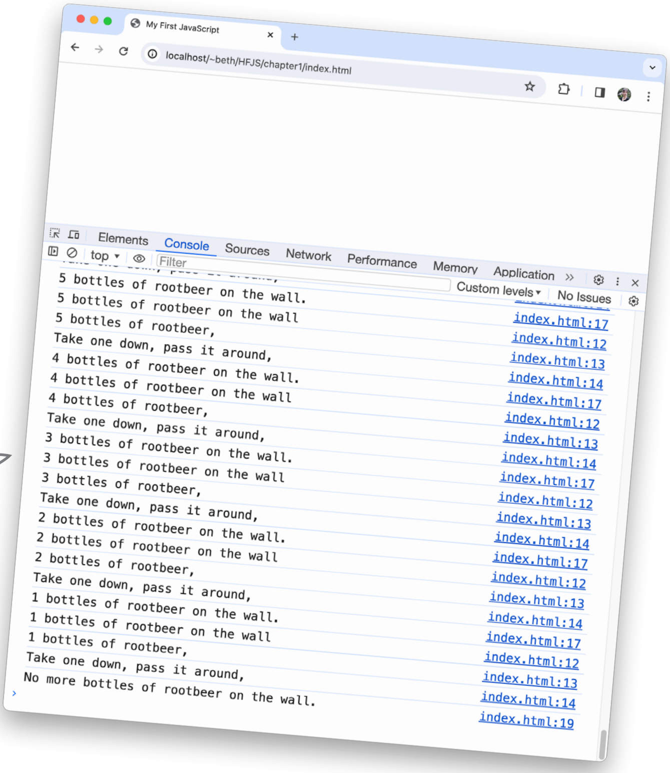
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script src="code.js">

    </script>
  </body>
</html>
```

Use the src attribute of the <script> element to link to your JavaScript file.

Where your code was.

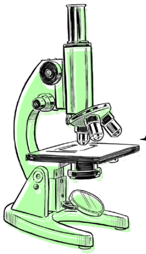
Believe it or not we still need the closing script tag, even if there is no code between the two tags.

④ That's it, the surgery is complete. Now you need to test it. Reload your "index.html" page, and you should see exactly the same result as before. Note that by using src="code.js", we're assuming that the code file is in the same directory as the HTML file.



```
5 bottles of rootbeer on the wall.        index.html:17
5 bottles of rootbeer on the wall         index.html:12
5 bottles of rootbeer,                    index.html:13
Take one down, pass it around,            index.html:14
4 bottles of rootbeer on the wall.        index.html:17
4 bottles of rootbeer on the wall         index.html:12
4 bottles of rootbeer,                    index.html:13
Take one down, pass it around,            index.html:14
3 bottles of rootbeer on the wall.        index.html:17
3 bottles of rootbeer on the wall         index.html:12
3 bottles of rootbeer,                    index.html:13
Take one down, pass it around,            index.html:14
2 bottles of rootbeer on the wall.        index.html:17
2 bottles of rootbeer on the wall         index.html:12
2 bottles of rootbeer,                    index.html:13
Take one down, pass it around,            index.html:14
1 bottles of rootbeer on the wall.        index.html:17
1 bottles of rootbeer on the wall         index.html:12
1 bottles of rootbeer,                    index.html:13
Take one down, pass it around,            index.html:14
No more bottles of rootbeer on the wall.  index.html:19
```

You should get the same result as before. But now your HTML and JavaScript are in separate files. Doesn't that just feel cleaner, more manageable, more stress-free already?



Anatomy of a Script Element

You know how to use the <script> element to add code to your page, but just to really nail down the topic, let's review the <script> element to make sure we have every detail covered:

The type attribute tells the browser you're writing JavaScript. The thing is, browsers assume you're using JavaScript if you leave it off. So, we recommend you leave it off, and so do the people who write the standards.

The <script> opening tag.

Don't forget the right bracket on the opening tag.

```
<script type="text/javascript">
    alert("Hello world!");
</script>
```

Everything between the script tags must be valid JavaScript.

You must end the script with a closing </script> tag, always!

When you are referencing a separate JavaScript file from your HTML, you'll use the <script> element like this:

Add a src attribute to specify the URL of the JavaScript file.

```
<script src="myJavaScript.js">
</script>
```

Use ".js" as the extension on JavaScript files.

Again, don't forget the closing </script> tag. You need it even when you're linking to an external file.

When referencing a separate JavaScript file, you don't put any JavaScript in the content of the <script> element.

You can't use inline and external together.

If you try throwing some quick code in between those <script> tags when you're already using a src attribute, it won't work. You'll need two separate <script> elements.

```
<script src="goodies.js">
  let x = "quick hack";
</script>
```

WRONG

JavaScript Exposed

This week's interview: Getting to know JavaScript

Head First: Welcome, JavaScript. We know you're super busy out there, working on all those web pages, so we're glad you could take time out to talk to us.

JavaScript: No problem. And, I *am* busier than ever these days; people are using JavaScript on just about every page on the web, for everything from simple menu effects to full-blown games. It's nuts!

Head First: That's amazing, given that just a few years ago, someone said that you were just a "half-baked scripting language." Now you're everywhere.

JavaScript: Don't remind me. I've come a long way since then, and many great minds have been hard at work making me better.

Head First: Better how? Seems like your basic language features are about the same…

JavaScript: Well, I'm better in a couple of ways. First of all, I'm lightning fast these days. While I'm considered a scripting language, now my performance is close to that of compiled languages.

Head First: And second?

JavaScript: My ability to do things in the browser has expanded dramatically. Using the JavaScript libraries available in all modern browsers, you can find out your location, play video and audio, paint graphics on your web page, and a lot more. But if you wanna do all that, you have to know JavaScript.

Head First: But back to those criticisms of you, the language. I've heard some not-so-kind words…I believe the phrase was "hacked-up language."

JavaScript: I'll stand on my record. I'm pretty much one of, if not *the* most widely used languages in the world. I've also fought off many competitors and won. Remember Java in the browser? Ha, what a joke. VBScript? Ha. JScript? Flash?! Silverlight? I could go on and on. So, tell me, how bad could I be?

Head First: You've been criticized as, well, "simplistic."

JavaScript: Honestly, it's my greatest strength. The fact that you can fire up a browser, type in a few lines of JavaScript and be off and running, that's powerful. And it's great for beginners too. I've heard some say there's no better beginning language than JavaScript.

Head First: But simplicity comes at a cost, no?

JavaScript: Well, that's the great thing. I'm simple in the sense you can get a quick start. But I'm deep and full of all the latest modern programming constructs.

Head First: Oh, like what?

JavaScript: Well, for example, can you say dynamic types, first-class functions, and closures?

Head First: I can say it, but I don't know what they are.

JavaScript: Figures…that's okay, if you stay with the book you will get to know them.

Head First: Well, give us the gist.

JavaScript: Let me just say this: JavaScript was built to live in a dynamic web environment—an exciting environment where users interact with a page, where data is coming in on the fly, where many types of events happen—and the language reflects that style of programming. You'll get it a little more a bit later in the book when you understand JavaScript more.

Head First: Okay, to hear you tell it, you're the perfect language. Is that right?

JavaScript tears up...

JavaScript: You know, I didn't grow up within the ivy-covered walls of academia like most languages. I was born into the real world and had to sink or swim very fast in my life. Given that, I'm not perfect; I certainly have a few "rough spots."

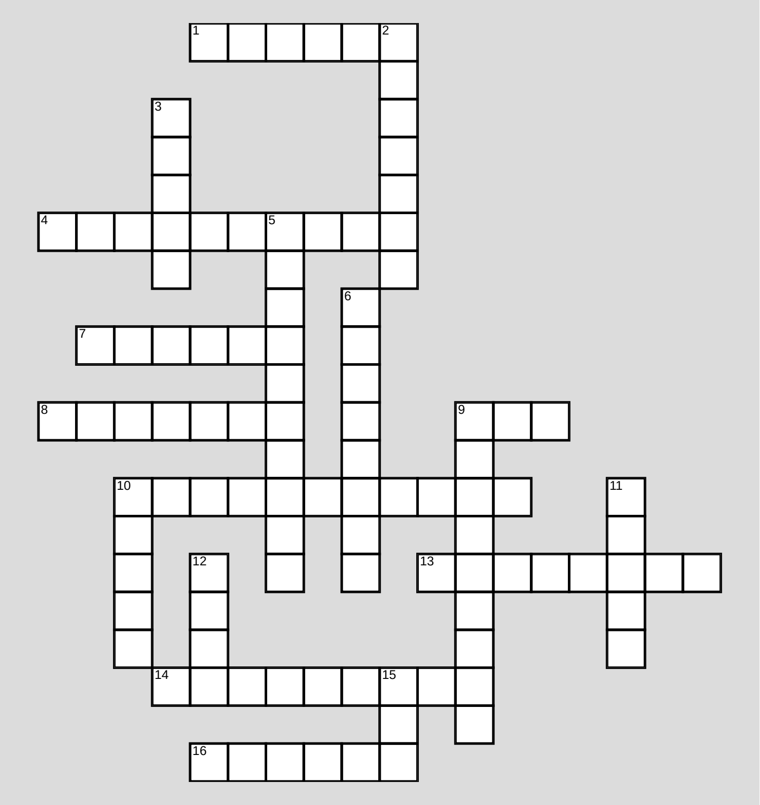
Head First, with a slight Barbara Walters smile: We've seen a new side of you today. I think this merits another interview in the future.

Bullet Points

- JavaScript is used to add **behavior** to web pages.
- JavaScript executes fast in modern browsers.
- Browsers begin executing JavaScript code as soon as they encounter the code in the page.
- Add JavaScript to your page with the **<script>** element.
- You can put your JavaScript inline in the web page, or link to a separate file containing your JavaScript from your HTML.
- Use the **src** attribute in the <script> tag to link to a separate JavaScript file.
- HTML **declares** the structure and content of your page; JavaScript **computes** values and adds behavior to your page.
- JavaScript programs are made up of a series of **statements**.
- One of the most common JavaScript statements is a variable declaration, which uses the **let** keyword to declare a new variable and the assignment operator, **=,** to assign a value to it.
- Use **const** to assign a value that shouldn't change.
- The value of a constant doesn't vary, so we call them constants, not variables.
- There are just a few rules and guidelines for naming JavaScript variables and constants, and it's important that you follow them.
- Remember to avoid JavaScript keywords when naming variables.
- JavaScript **expressions** compute values.
- Three common types of expressions are **numeric**, **string**, and **boolean** expressions.
- **if/else** statements allow you to make decisions in your code.
- **while/for** statements allow you to execute code many times by looping.
- You can group statements together into a code **block** by enclosing them in curly braces.
- Use **console.log** instead of **alert** to display messages to the console.
- Console messages should be used primarily for troubleshooting, as users will most likely never see them.
- JavaScript is most commonly found adding behavior to web pages, but it's also used to script many creative applications and is used as a server-side programming language with Node.js.

JavaScript cross

Time to stretch your dendrites with a puzzle to help it all sink in.



ACROSS

1. Variables are used to store these.
4. Use _______ to troubleshoot your code.
7. Today's JavaScript runs a lot _______ than it used to.
8. There are 99 _______ of rootbeer on the wall.
9. To link to an external JavaScript file from HTML, you need the _______ attribute for your <script> element.
10. Each time through a loop, we evaluate a _______ expression.
13. The if/else statement is used to make a _______.
14. All JavaScript statements end with a _______.
16. You put your JavaScript inside a _______ element.

DOWN

2. You can concatenate _______ together with the + operator.
3. Store values that don't change in this.
5. 3 + 4 is an example of an _______.
6. JavaScript adds _______ to your web pages.
9. Each line of JavaScript code is called a _______.
10. To avoid embarrassing naming mistakes, use _______ case.
11. Do things more than once in a JavaScript program with the _______ loop.
12. JavaScript variable names are _______ sensitive.
15. To declare a variable, use this keyword.

→ Solution on page 45

Sharpen your pencil Solution

From page 13

Identify the declarations below that you think are best suited for let and for const. Here's our solution.

const `DISTANCE_TO_MOON = 238900;`

The distance to the moon isn't going to change during this computation, so this is a constant value. At least, we hope it doesn't change!

let `last_guess = 0;`

Presumably this is to keep the value of a user guess that is going to change over time, so this is not a constant and we use let.

const `SCREEN_WIDTH = 1024;`

Although screen sizes can sometimes change, like if the browser window resizes, in this case the uppercase treatment signifies that this is intended to be a fixed constant value.

let `counter = 100;`

If we're counting something, then this value needs to change over time; thus, we have a variable and use let.

const `firstUSPresident = "Washington";`

The first president of the United States will always be George Washington, so this is a const.

let `fluxCapacitorReading = "System Normal";`

You never know when the status of the flux capacitor is going to change. Sure, everything may be normal now, but watch out! So, we have a variable declared with let.

From page 16

BE the Browser Solution

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. Here's our solution.

A

```
// Test for jokes
const joke = "JavaScript walked into a bar....';
let toldJoke = "false";
let $punchline =
  "Better watch out for those semi-colons."
let %entage = 20;
let result

if (toldJoke == true) {
    Alert($punchline);
} else
    alert(joke);
}
joke = "Knock, knock. Who's there? JavaScript....";
```

Delimit your strings with two double quotes (") or two single quotes ('). Don't mix!

Don't put quotes around boolean values unless you really want a string.

It's okay, but not recommended, to begin a variable name with a $.

Don't forget to end statements with a semicolon!

Can't use % in variable names.

Another missing semicolon.

Should be alert, not Alert. JavaScript is case sensitive.

We're missing an opening brace here.

We're trying to change the value of a constant here.

B

```
\\ Movie Night
let zip code = 98104;
const joe'sFavoriteMovie = Forbidden Planet;
let movieTicket$    =    9;

if (movieTicket$ >= 9) {
    alert("Too much!");
} else {
    alert("We're going to see " + joe'sFavoriteMovie);
}
```

Comments should begin with // not \\.

No spaces allowed in variable names.

No quotes allowed in variable names.

But we do need quotes around the string "Forbidden Planet".

This if/else doesn't work because of the invalid constant name here.

Sharpen your pencil Solution

From page 18

Get out your pencil and let's put some expressions through their paces. For each expression below, compute its value and write in your answer. Yes, WRITE IN...forget what your Mom told you about writing in books and scribble your answer right in this book! Here's our solution.

Can you say "Celsius to Fahrenheit calculator"?

`(9 / 5) * temp + 32`

What is the result when temp is 10? 50

This is a boolean expression. The == operator tests if two values are equal to each other.

`color == "orange"`

Is this expression true or false when color has the value "pink"? false
Or has the value "orange"? true

`name + ", " + "you've won!"`

What value does this compute to when name is "Martha"?
"Martha, you've won!"

This tests if the first value is greater than the second. You can also use >= to test if the first value is greater than or equal to the second.

`yourLevel > 5`

When yourLevel is 2, what does this evaluate to? false
When yourLevel is 5, what does this evaluate to? false
When yourLevel is 7, what does this evaluate to? true

`(level * points) + bonus`

Okay, level is 5, points is 30,000, and bonus is 3,300. What does this evaluate to? 153300

`color != "orange"`

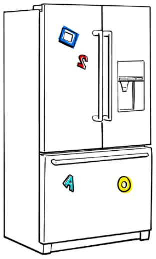
Is this expression true or false when color has the value "pink"? true

The != operator tests if two values are NOT equal to each other.

Extra CREDIT! We explain why this happens in a later chapter.

`1000 + "108"`

Are there a few possible answers? Only one is correct. Which would you choose? "1000108"



Code Magnets Solution

From page 26

A JavaScript program is all scrambled up on the fridge. Can you put the magnets back in the right places to make a working program to produce the output shown below? Here's our solution.

Here are the unscrambled magnets!

```
const name = "Joe";
```

Declare a constant for the name "Joe". The name doesn't change, so we use const, not let.

```
let i = 0;
```

i is our while loop variable; it needs to change each time through the loop.

```
while (i < 2) {
```

Keep looping as long as the value in i is less than 2.

```
    console.log("Happy Birthday to you.");
```

Display a message to the browser page.

```
    i = i + 1;
```

Increment the while loop variable.

```
}
```

End the while loop block.

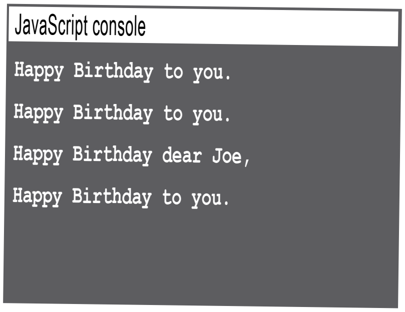
```
console.log("Happy Birthday dear " + name);
```

Display a message to the browser page, including the value of the variable name.

```
console.log("Happy Birthday to you.");
```

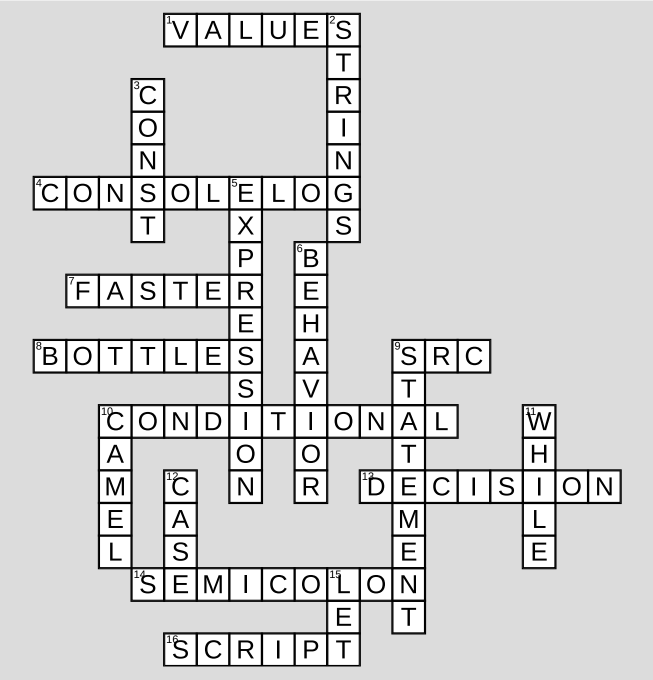
Display a message to the browser page.

Your unscrambled program should produce this output in the console.



```
Happy Birthday to you.
Happy Birthday to you.
Happy Birthday dear Joe,
Happy Birthday to you.
```

JavaScript cross Solution

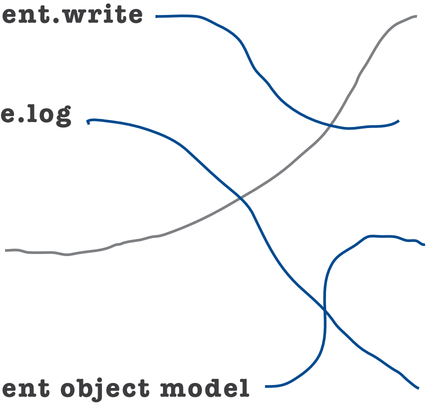

1 VALUES
2 STRINGS
3 CONST
4 CONSOLELOG
5 EXPRESSION
6 BEHAVIOR
7 FASTER
8 BOTTLES
9 SRC
9 STATEMENT
10 CONDITIONAL
10 CAMEL
11 WHILE
12 CASE
13 DECISION
14 SEMICOLON
15 LET
16 SCRIPT

From page 40

Who Does What? Solution

From page 28

All our methods of communication have come to the party with masks on. Can you help us unmask each one? Match the descriptions on the right to the names on the left. Here's our solution.



document.write

console.log

alert

document object model

I'll stop your user in her tracks and deliver a short message. The user has to click "OK" to go further.

I can insert a little HTML and text into a document. I'm not the most elegant way to get a message to your users, and I'm being phased out of browsers.

Using me, you can totally control a web page: get values that a user typed in, alter the HTML or the style, or update the content of your page.

I'm just here for simple debugging purposes. Use me and I can write out information to a special developer's console.

2 writing real code

Going further

You already know about variables, types, expressions... we could go on. The point is, you already know a few things about JavaScript. In fact, you know enough to write some **real code**. Some code that does something interesting, some code that someone would want to use. What you're lacking is the **real experience** of writing code, and we're going to remedy that right here and now. How? By jumping in head first and coding up a casual game, all written in JavaScript. Our goal is ambitious, but we're going to take it one step at a time. Come on, let's get this started, and if you want to launch the next startup, we won't stand in your way; the code is yours.

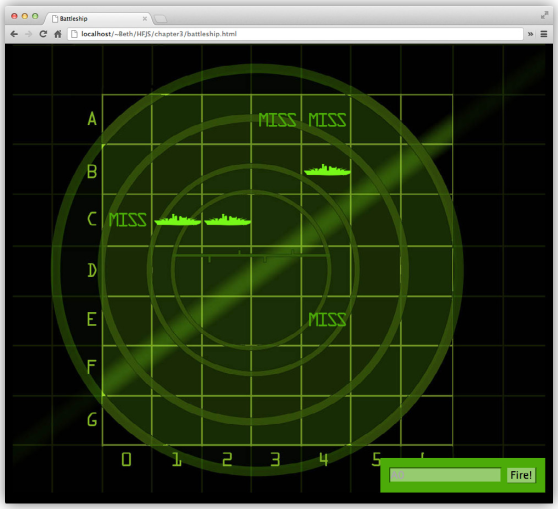
Let's build a Battleship game

It's you against the browser: the browser hides ships and your job is to seek them out and destroy them. Of course, unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

Goal: Sink the browser's ships in the fewest number of guesses. You're given a rating, based on how well you perform.

Setup: When the game program is launched, the computer places ships on a virtual grid. When that's done, the game asks for your first guess.

How you play: The browser will prompt you to enter a guess, and you'll type in a grid location. In response to your guess, you'll see a result of "HIT!" "MISS" or "You sank my battleship!" When you sink all the ships, the game ends by displaying your rating.



Here's what we're shooting for: a nice 7x7 grid with three ships to hunt down. Right now we're going to start a little simpler, but once you know a bit more JavaScript we'll complete the implementation so it looks just like this, complete with graphics and everything...we'll leave the sound to you as extra credit.

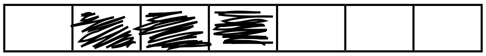
Our first attempt...

...a simplified Battleship

For our first attempt, we're going to start simpler than the full-blown 7 by 7 graphical version with three ships. Instead, we're going to start with a nice 1D grid with seven locations and one ship to find. It will be crude, but our focus is on designing the basic code for the game, not the look and feel (at least for now).

Don't worry; by starting with a simplified version of the game, you get a big head start on building the full game later. This also gives us a nice chunk to bite off for your first real JavaScript program (not counting the serious business application from Chapter 1—99 bottles of rootbeer on the wall—of course). So, we'll build the simple version of the game in this chapter, and we'll get to the deluxe version later in the book, after you've learned a bit more about JavaScript.

Instead of a 7x7 grid, like the one above, we're going to start with just a 1x7 grid. And we'll worry about just one ship for now.



Notice that each ship takes up three grid locations (similar to the real board game).

First, a high-level design

We know we'll need variables, and some numbers and strings, and if statements, and conditional tests, and loops...but where, and how many? And how do we put all this together? To answer these questions, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:

1 User starts the game.

- **A** Game places a battleship at a random location on the grid.

2 Gameplay begins.

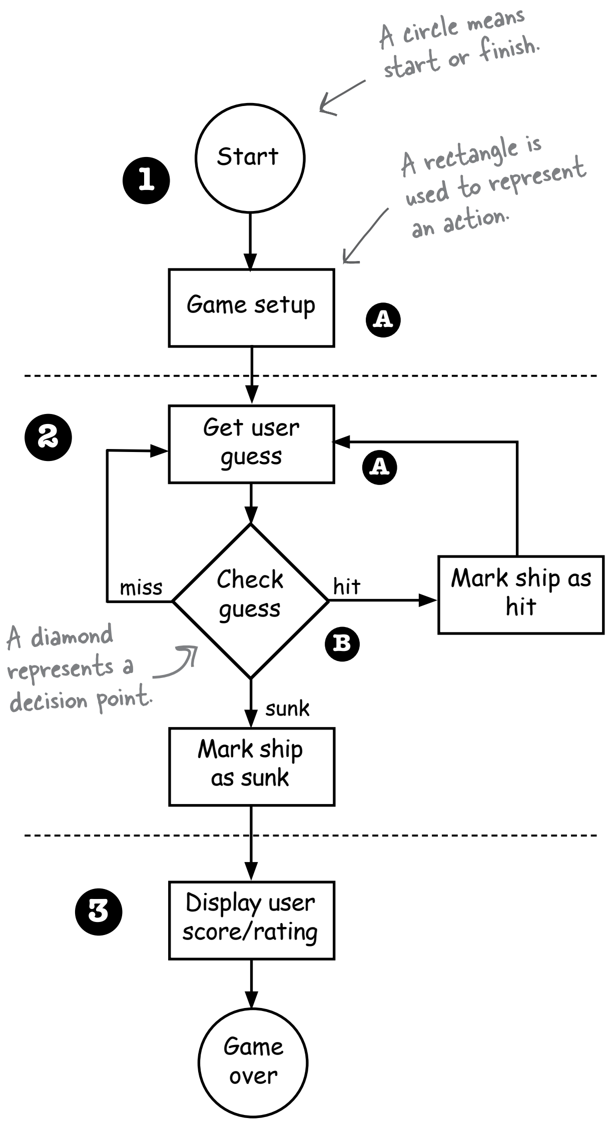
Repeat the following until the battleship is sunk:

- **A** Prompt user for a guess ("2", "0", etc.).
- **B** Check the user's guess against the battleship to look for a hit, miss, or sink.

3 Game finishes.

Give the user a rating based on the number of guesses.

Now we have a high-level idea of the kinds of things the program needs to do. Next, we'll figure out a few more details for the steps.


1
Start
Game setup
A
2
Get user guess
A
miss
Check guess
hit
Mark ship as hit
B
sunk
Mark ship as sunk
3
Display user score/rating
Game over

A circle means start or finish.

A rectangle is used to represent an action.

A diamond represents a decision point.

Whoa. A real flowchart.

A few more details...

We have a pretty good idea about how this game is going to work from the high-level design and professional-looking flowchart, but let's nail down just a few more of the details before we begin writing the code.

Representing the ships

For one thing, we can start by figuring out how to represent a ship in our grid. Keep in mind that the virtual grid is…well, *virtual*. In other words, it doesn't exist anywhere in the program. As long as both the game and the user know that the battleship is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code. You might be tempted to build something that holds all seven locations and then to try to place the ship in those locations. But we don't need to. We just need to know the cells where the ship is located, say, at cells 1, 2, and 3.

Getting user input

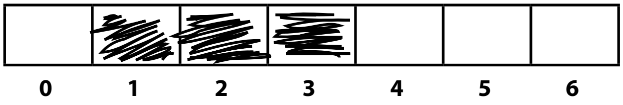
What about getting user input? We can do that with the `prompt` function. Whenever we need to get a new location from the user, we'll use `prompt` to display a message and get the input, which is just a number between 0 and 6, from the user.

Displaying the results

What about output? For now, we'll continue to use `alert` to show the output of the game. It's a bit clunky, but it'll work. (For the real game, later in the book, we'll be updating the web page instead, but we've got a way to go before we get there.)

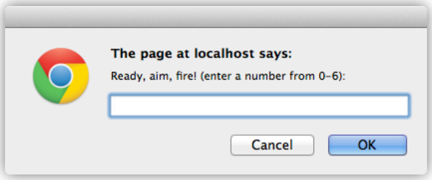
1 **Game starts**, creates one battleship, and gives it a location on three cells in the single row of seven cells.

The locations are just integers; for example, 1, 2, 3 are the cell locations in this picture:



2 **Gameplay begins**. Prompt user for a guess:

A

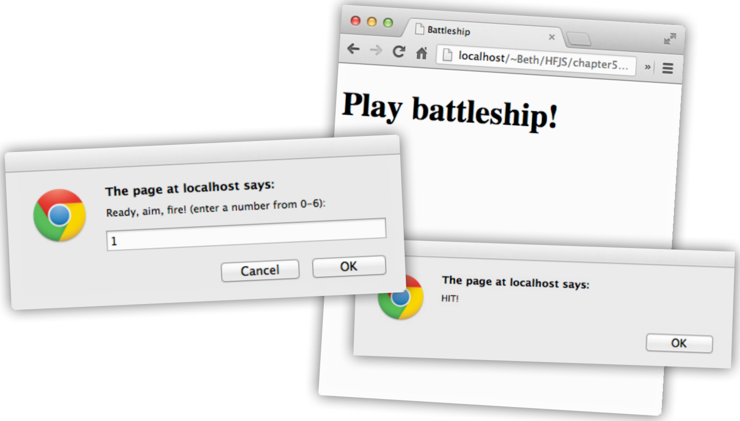

The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

B Check to see if user's input hit any of the ship's three cells. Keep track of how many hits there are in a variable.

3 **Game finishes** when all three cells have been hit and our number of hits variable's value is 3. We tell the user how many guesses it took to sink the ship.

Sample game interaction



Play battleship!

The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

1

The page at localhost says:

HIT!

Working through the pseudocode

We need an approach to planning and writing our code. We're going to start by writing *pseudocode.* Pseudocode is halfway between real JavaScript code and a plain English description of the program, and as you'll see, it will help us think through how the program is going to work without having to fully develop the *real code*.

In this pseudocode for Simple Battleship, we've included a section that describes the variables we'll need, and a section describing the logic of the program. The variables will tell us what we need to keep track of in our code, and the logic describes what the code has to faithfully implement to create the game.

The variables we need.

```
DECLARE three variables to hold the locations of each cell of the ship. Let's call them
location1, location2, and location3.

DECLARE a variable to hold the user's current guess. Let's call it guess.

DECLARE a variable to hold the number of hits. We'll call it hits and set it to 0.

DECLARE a variable to hold the number of guesses. We'll call it guesses and set it to 0.

DECLARE a variable to keep track of whether the ship is sunk or not. Let's call it isSunk
and set it to false.
```

And here's the logic.

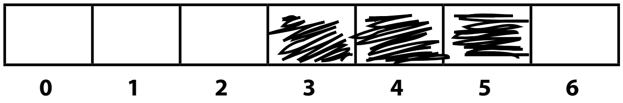
```
LOOP: while the ship is not sunk
    GET the user's guess
    COMPARE the user's input to valid input values
    IF the user's guess is invalid,
        TELL user to enter a valid number
    ELSE
        ADD one to guesses
        IF the user's guess matches a location
            ADD one to the number of hits
            IF number of hits is 3
                SET isSunk to true
                TELL user "You sank my battleship!"
            END IF
        END IF
    END ELSE
END LOOP
TELL user stats
```

It's not JavaScript, but you can probably already see how to begin implementing this logic in code.

Notice that we're using indentation to help make the pseudocode easier to read. We'll be doing that in the real code too.

Sharpen your pencil

Let's say our virtual grid looks like this:



And we've represented the ship's location using our location variables, like this:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Use the following sequence as your test user input:

```
1, 4, 2, 3, 5
```

Now, using the pseudocode on the previous page, walk through each step of the code and see how this works given the user input. Put your notes below. We've begun the exercise for you. If this is your first time walking through pseudocode, take your time and see how it all works.

If you need a hint, take a quick peek at our answer at the end of the chapter.

location1	location2	location3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false

The first row shows the initial values of the variables, before the user enters their first guess. We're not initializing the variable guess, so its value is undefined.

Solution on page 77

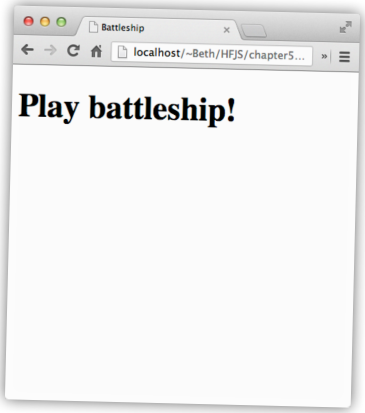
Oh, before we go any further, don't forget the HTML!

You're not going to get very far without some HTML to link to your code. Go ahead and type the markup below into a new file named "battleship.html". After you've done that, we'll get back to writing code.

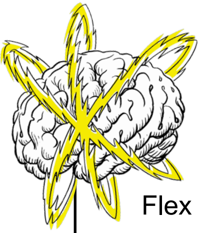
```
<!doctype html>
<html lang="en">
  <head>
    <title>Battleship</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Play battleship!</h1>
    <script src="battleship.js"></script>
  </body>
</html>
```

The HTML for the Battleship game is super simple; we just need a page that links to the JavaScript code, and that's where all the action happens.

We're linking to the JavaScript at the bottom of the <body> of the page, so the page is loaded by the time the browser starts executing the code in "battleship.js".


Play battleship!

Here's what you'll see when you load the page. We need to write some code to get the game going!



Brain Power

Flex those dendrites.

This is thinking ahead a bit, but what kind of code do you think it would take to generate a random location for the ship each time you load the page? What factors would you have to take into account in the code to correctly place a ship? Feel free to scribble some ideas here.

Writing the Simple Battleship code

We're going to use the pseudocode as a blueprint for our real JavaScript code. First, let's tackle all the variables we need. Take another look at our pseudocode:

```
DECLARE three variables to hold the locations of each cell of the ship. Let's call them location1, location2, and location3.
DECLARE a variable to hold the user's current guess. Let's call it guess.
DECLARE a variable to hold the number of hits. We'll call it hits and set it to 0.
DECLARE a variable to hold the number of guesses. We'll call it guesses and set it to 0.
DECLARE a variable to keep track of whether the ship is sunk or not. Let's call it isSunk and set it to false.
```

We need three variables to hold the ship's location.

And three more (guess, hits, and guesses) to deal with the user's guess.

And another to track whether or not the ship is sunk.

Let's get these variables into a JavaScript file. Create a new file named "battleship.js" in the same folder as "battleship.html" and type in your variable declarations like this:

```
let location1 = 3;
let location2 = 4;
let location3 = 5;


let guess;
let hits = 0;
let guesses = 0;


let isSunk = false;
```

Here are our three location variables. We'll go ahead and set up a ship at locations 3, 4, and 5, just for now.

We'll come back later and write some code to generate a random location for the ship to make it harder for the user.

We didn't make these constants because we'll vary them later.

The variable guess won't have a value until the user makes a guess. Until then, it will have the value undefined.

We'll assign initial values of 0 to both hits and guesses.

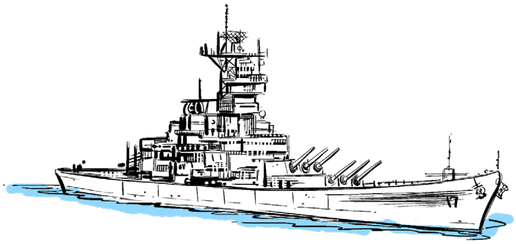
Finally, the isSunk variable gets a value of false. We'll set this to true when we've sunk the ship.

Serious Coding

If you don't provide an initial value for a variable, then JavaScript gives it a default value of `undefined`. Think of the value `undefined` as JavaScript's way of saying "this variable hasn't been given a value yet." We'll be talking more about undefined and some other strange values a little later.

Now let's write the game logic

We've got the variables out of the way, so let's dig into the actual pseudocode that implements the game. We'll break this into a few pieces. The first thing you're going to want to do is implement the loop: it needs to keep looping while the ship isn't sunk. From there, we'll take care of getting the guess from the user and validating it—you know, making sure it really is a number between 0 and 6—and then we'll write the logic to check for a hit on the ship and to see if the ship is sunk. Finally, we'll create a little report for the user with the number of guesses it took to sink the ship.



```
LOOP: while the ship is not sunk
    GET the user's guess
    COMPARE the user's input to valid input values
    IF the user's guess is invalid,
        TELL user to enter a valid number
    ELSE
        ADD one to guesses
        IF the user's guess matches a location
            ADD one to the number of hits
            IF number of hits is 3
                SET isSunk to true
                TELL user "You sank my battleship!"
            END IF
        END IF
    END ELSE
END LOOP
TELL user stats
```

Step 1: Set up the loop, get the user's input, and validate it.

Step 2: Check the guess. Is it a hit? A miss?

Step 3: Check to see if the ship is sunk.

Step 4: Handle the final message to the user.

To Do:

- ☐ Create loop and get user guess
- ☐ Check user guess
- ☐ Check if ship has been sunk
- ☐ Display stats to user

Step 1: Setting up the loop, getting some input

☐ Create loop and get user guess
☐ Check user guess
☐ Check if ship has been sunk
☐ Display stats to user

Now we're going to begin to translate the logic of our game into actual JavaScript code. There isn't a perfect mapping from pseudocode to JavaScript, so you'll see a few adjustments here and there. The pseudocode gives us a good idea of *what* the code needs to do, and now we have to write the JavaScript code that can do the *how*.

Let's start with all the code we have so far and then we'll zero in on just the parts we're adding (to save a few trees here and there, or electrons if you're reading the digital version of the book):

DECLARE variables

```
let location1 = 3;
let location2 = 4;
let location3 = 5;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;
```

We've already covered these, but we're including them here for completeness.

LOOP: *while the ship is not sunk*

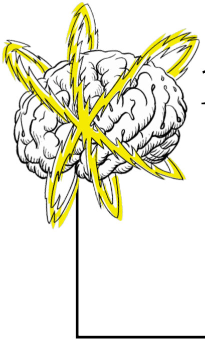
GET the user's guess

```
while (isSunk == false) {

    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");

}
```

Here's the start of the loop. While the ship isn't sunk, we're still in the game, so keep looping.

Remember, while uses a conditional test to determine whether to keep looping. In this case we're testing to make sure that isSunk is still false. We'll set it to true as soon as the ship is sunk.

Each time we go through the while loop, we're going to ask the user for a guess. To do that, we use the prompt built-in function. More on that on the next page...



Brain Power

If you ran this code now, would the game ever end?

How prompt works

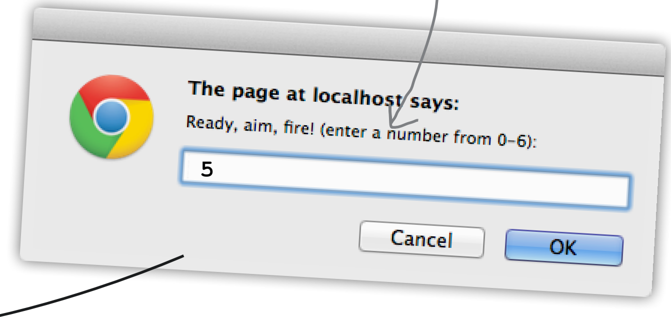
The browser provides a built-in function you can use to get input from the user, named `prompt`. The `prompt` function is a lot like the `alert` function you've already used—`prompt` causes a dialog to be displayed with a string that you provide, just like `alert`—but it also gives the user a place to type a response. That response, in the form of a string, is then returned as a result of calling the function. Now, if the user cancels the dialog, then `null` is returned instead.

Here we're assigning the result of the prompt function to the guess variable.

```
guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```

You provide prompt with a string, which is used as instructions to your user in the dialog box.

The prompt function's job is to get input from the user. Depending on your device, that usually happens in a dialog box.



The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

5

"5"

Once the prompt function obtains input from the user, it returns that input to your code. In this case the input, in the form of a string, is assigned to the variable guess.

Watch it!

You might be tempted to try this code now...

...but don't. If you do, your browser will start an *infinite loop* of asking you for a guess, and then asking you for a guess, and so on, without any means of stopping the loop (other than using your operating system to force the browser process to stop).

Step 2: Checking the user's guess

☑ Create loop and get user guess
☐ Check user guess
☐ Check if ship has been sunk
☐ Display stats to user

If you look at the pseudocode, you'll see that to check the user's guess, we need to first make sure the user has entered a valid input. If so, then we also check to see if the guess was a hit or a miss. And, we'll want to make sure we appropriately update the `guesses` and `hits` variables. Let's get started by checking the validity of the user's input, and, if the input is valid, we'll increment the `guesses` variable. After that, we'll write the code to see if the user has a hit or a miss.

```
// Variable declarations go here


while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
       alert("Please enter a valid cell number!");
    } else {
       guesses = guesses + 1;
    }
}
```

We check validity by making sure the guess is between zero and six.

If the guess isn't valid, we'll tell the user with an alert.

And if the guess is valid, go ahead and add one to guesses so we can keep track of how many times the user has guessed.

Let's look a little more closely at the validity test. You know we're checking to see that the guess is between zero and six, but how exactly does this conditional test that? Let's break it down:

Try to read this like it's English: this conditional is true if the user's guess is less than zero OR the user's guess is greater than six. If either is true, then the input is invalid.

```
if (guess < 0 || guess > 6) {
```

This is really just two small tests put together. The first test checks if guess is less than zero.

And this one checks to see if guess is greater than six.

And this, which we call the OR operator, combines the two tests so that if either test is true, then the entire conditional is true. If both tests are false, then the statement is false, and the guess is between zero and six, which means it's valid.

there are no Dumb Questions

Q: I noticed there is a Cancel button on the prompt dialog box. What gets returned from the prompt function if the user hits Cancel?

A: If you click Cancel in the prompt dialog box, then prompt returns the value null rather than a string. null means "no value," which is appropriate in this case because you've canceled without entering a value. We can use the fact that the value returned from prompt is null to check to see if the user clicked Cancel, and if they did, then we could, say, end the game. We're not doing that in our code, but keep this idea in the back of your mind, as we might use it later in the book.

Q: You said that prompt always returns a string. So how can we compare a string value, like "0" or "6", to numbers, like 0 and 6?

A: In this situation, JavaScript tries to convert the string in guess to a number in order to do the comparisons, guess < 0 and guess > 6. As long as you enter only a number, like 4, JavaScript knows how to convert the string "4" to the number 4 when it needs to. We'll come back to the topic of type conversion in more detail later.

Q: What happens if the user enters something that isn't a number into the prompt? Like "six" or "quit"?

A: In that case, JavaScript won't be able to convert the string to a number for the comparison. So, you'd be comparing "six" to 6 or "quit" to 6, and that kind of comparison will return false, which will lead to a MISS. In a more robust version of Battleship, we'll check the user input more carefully and make sure they've entered a number first.

Q: With the OR operator, is it true if only one or the other is true, or can both be true?

A: Yes, both can be true. The result of the OR operator (||) is true if either of the tests is true, or if both are true. If both are false, then the result is false.

Q: Is there an AND operator?

A: Yes! The AND operator (&&) works similarly to OR, except that the result of AND is true only if both tests are true.

Q: What's an infinite loop?

A: Great question. An infinite loop is one of the many problems that plague programmers. Remember that a loop requires a conditional test, and the loop will continue as long as that conditional test is true. If your code never does anything to change things so that the conditional test is false at some point, the loop will continue forever. And ever. Until you kill your browser or reboot.

Two-Minute Guide to Boolean Operators

A boolean operator is used in a boolean expression, which results in a true or false value. There are two kinds of boolean operators: comparison operators and logical operators.

Comparison Operators

Comparison operators compare two values. Here are some common comparison operators:

<	means "less than"
>	means "greater than"
==	means "equal to"
===	means "exactly equal to" (we'll come back to this one later)
<=	means "less than or equal to"
>=	means "greater than or equal to"
!=	means "not equal to"

Logical Operators

Logical operators combine two boolean expressions to create one boolean result (true or false). Here are two logical operators:

||	means OR. Results in true if *either* of the two expressions is true.
&&	means AND. Results in true if *both* of the two expressions are true.

Another logical operator is NOT, which acts on one boolean expression (rather than two):

!	means NOT. Results in true if the expression evaluates to false.

So, do we have a hit?

☑ Create loop and get user guess
☐ Check user guess
☐ Check if ship has been sunk
☐ Display stats to user

This is where things get interesting—the user's taken a guess at the ship's location, and we need to write the code to determine if that guess has hit the ship. More specifically, we need to see if the guess matches one of the locations of the ship. If it does, then we'll increment the `hits` variable.

Here's a first stab at writing the code for the hit detection—let's step through it:

```
if (guess == location1) {
     hits = hits + 1;
} else  if (guess == location2) {
     hits = hits + 1;
} else if (guess == location3) {
     hits = hits + 1;
}
```

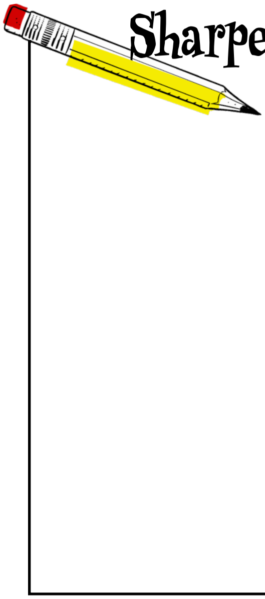
If the guess is location1, then we hit the ship, so increment the hits variable by one.

Otherwise, if the guess is location2, then do the same thing.

Finally, if the guess is location3, we increment the hits variable.

And if none of these are true, then the hits variable is never incremented.

Notice we're using indentation for the code in each if/else block. This makes your code easier to read, especially when you've got lots of blocks nested inside blocks.



Sharpen your pencil

What do you think of this first attempt to write the code to detect when a ship is hit? Does it look more complex than it needs to be? Are we repeating code in a way that seems a bit, well, redundant? Could we simplify it? Using what you know of the || operator (that is, the boolean OR operator), can you simplify this code? Make sure you check your answer at the end of the chapter before moving on.

→ Solution on page 81

Adding the hit detection code

☑ Create loop and get user guess
☑ Check user guess
☐ Check if ship has been sunk
☐ Display stats to user

Let's put everything together from the previous couple of pages:

```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      alert("Please enter a valid cell number!");
    } else {
      guesses = guesses + 1;

      if (guess == location1 || guess == location2 || guess == location3) {
          hits = hits + 1;
      }
    }
}
```

LOOP: *while the ship is not sunk*

GET the user's guess

ADD one to guesses

IF the user's guess matches a location

ADD one to the number of hits

Check the user's guess...

The user's guess looks valid, so let's increase the number of guesses by one.

If the guess matches one of the ship's locations, we increment the hits counter.

We've combined the three conditionals into one if statement using || (OR). So read it like this: "If guess is equal to location1 OR guess is equal to location2 OR guess is equal to location3, increment hits."

Step 3: Hey, you sank my battleship!

☑ Create loop and get user guess
☑ Check user guess
☑ Check if ship has been sunk
☐ Display stats to user

We're nearly there; we've almost got this game logic nailed down. Looking at the pseudocode again, what we need to do now is test to see if we have three hits. If we do, then we've sunk a battleship. And if we've sunk a battleship, then we need to set `isSunk` to true and also tell the user they've destroyed a ship. Let's sketch out the code again before adding it in:

First, check to see if there are three hits.

```
if (hits == 3) {
    isSunk = true;
    alert("You sank my battleship!");
}
```

If so, set isSunk to true.

Take another look at the while loop above. What happens when isSunk is true?

And also let the user know.

Step 4: Provide some post-game analysis

☑ Create loop and get user guess
☑ Check user guess
☑ Check if ship has been sunk
☑ Display stats to user

After `isSunk` is set to true, the while loop is going to stop looping. That's right, this program we've come to know so well is going to stop executing the body of the while loop, and before you know it the game's going to be over. But we still owe the user some stats on how they did. Here's some code that does that:

```
let stats = "You took " + guesses + " guesses to sink the battleship, " +
                "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Here we're creating a string that contains a message to the user including the number of guesses they took, along with the accuracy of their shots. Notice that we're splitting up the string into pieces (to insert the variable guesses, and also to fit the string into multiple lines) using the concatenation operator, +. For now just type this as is, and we'll explain more about this later.

Now let's add this and the sunk ship detection into the rest of the code:

```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;

            if (hits == 3) {

                isSunk = true;

                alert("You sank my battleship!");
            }
        }
    }
}
let stats = "You took " + guesses + " guesses to sink the battleship, " +
                "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

LOOP: *while the ship is not sunk*

GET the user's guess

ADD one to guesses

IF the user's guess matches a location

ADD one to the number of hits

IF number of hits is 3

SET isSunk to true

TELL user "You sank my battleship!"

TELL user stats

Exercise

Remember we said pseudocode often isn’t perfect? Well, we actually left something out of our original pseudocode: we’re not telling the user if her guess is a HIT or a MISS. Can you insert these pieces of code in the proper place to correct this?

`alert("HIT!");`

`else {`

`}`

`alert("MISS");`

Here’s the code you’ll need to insert.

```
// Variable declarations go here


while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;
            if (hits == 3) {
              isSunk = true;
              alert("You sank my battleship!");
            }
        }
    }
}
let stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

This is a lot of curly braces to match. If you’re having trouble matching them, just draw lines right in the book to match them up.

Solution on page 80

And that completes the logic!

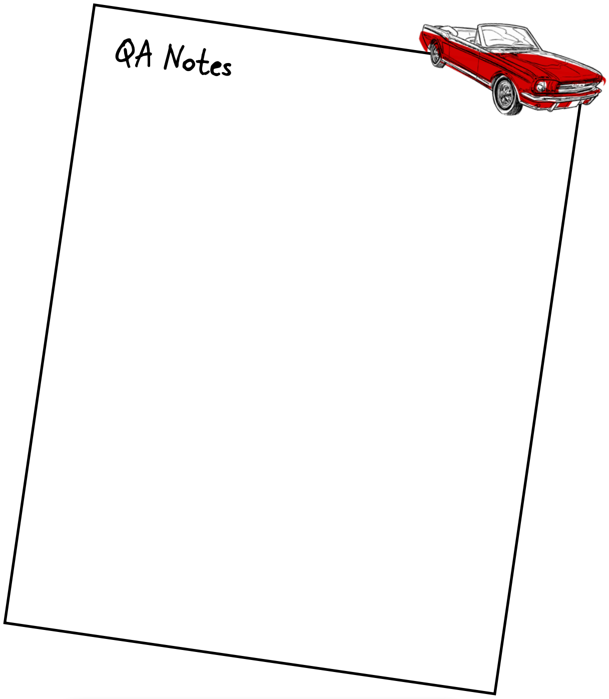
☑ Create loop and get user guess
☑ Check user guess
☑ Check if ship has been sunk
☑ Display stats to user

Alright! We've now fully translated the pseudocode to actual JavaScript code. We even discovered something we left out of the pseudocode, and we've got that accounted for too. Below, you'll find the code in its entirety. Make sure you have this typed in and saved in "battleship.js":

```
let location1 = 3;
let location2 = 4;
let location3 = 5;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        } else {
           alert("MISS");
        }
    }
}
let stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

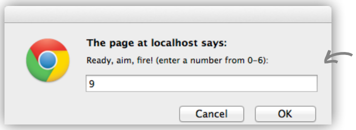
Doing a little quality assurance

Quality assurance, or QA, is the process of testing software to find defects. We're going to do a little QA on our code. When you're ready, load "battleship.html" in your browser and start playing. Try some different things. Is it working perfectly? Or did you find some issues? If so, list them here. You can see our test run on this page too.

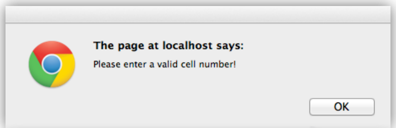


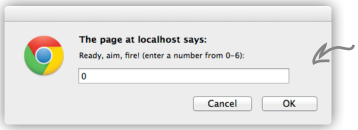
Jot down anything that doesn't work the way it should, or that could be improved.

Here's what our game interaction looked like.

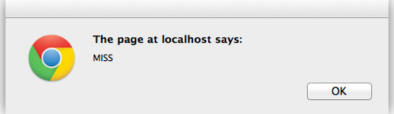

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
9

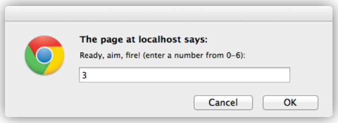
First we enter an invalid number, 9.


The page at localhost says:
Please enter a valid cell number!

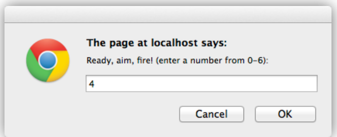

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
0

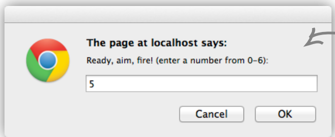
Then we enter 0, to get a miss.

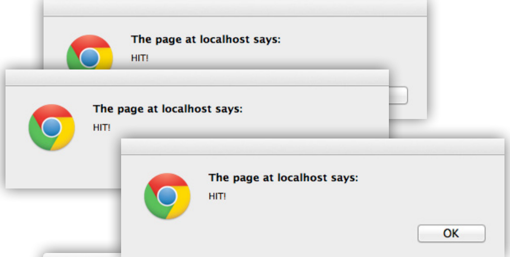

The page at localhost says:
MISS


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
3

But then we get three hits in a row!


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
4

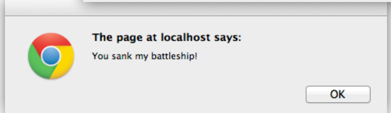

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
5


The page at localhost says:
HIT!

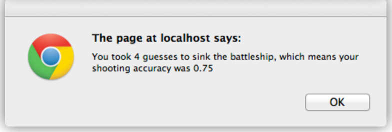
The page at localhost says:
HIT!

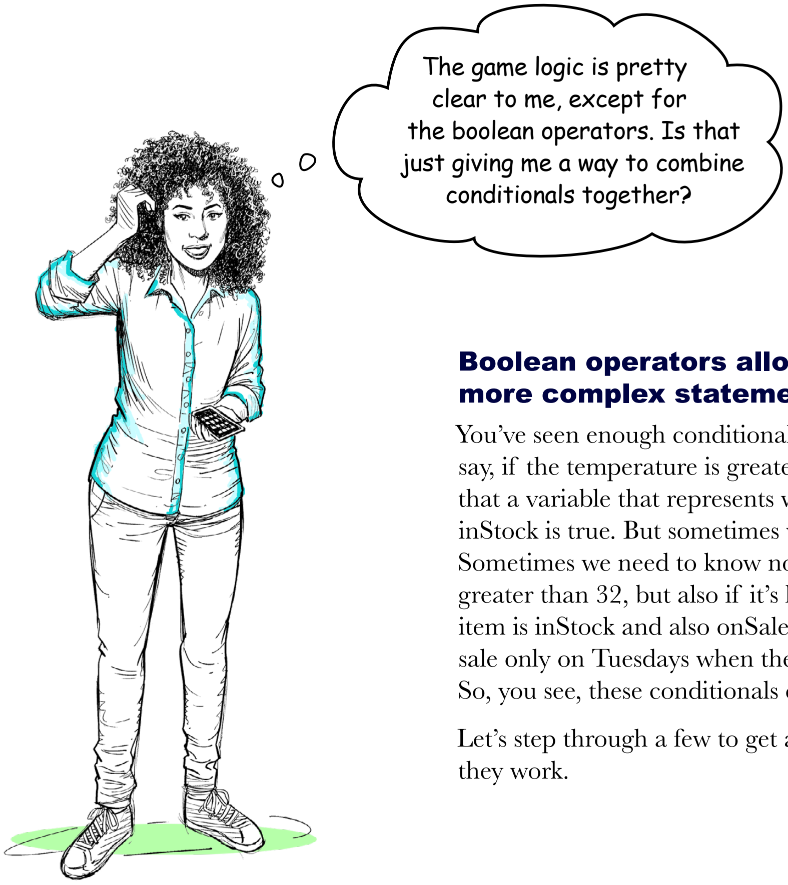
The page at localhost says:
HIT!

On the third and final hit, we sink the battleship.


The page at localhost says:
You sank my battleship!

And we see that it took 4 guesses to sink the ship, with an accuracy of 0.75.


The page at localhost says:
You took 4 guesses to sink the battleship, which means your shooting accuracy was 0.75


The game logic is pretty clear to me, except for the boolean operators. Is that just giving me a way to combine conditionals together?

Boolean operators allow you to write more complex statements of logic.

You've seen enough conditionals to know how to test, say, if the temperature is greater than 32 degrees. Or, that a variable that represents whether an item is inStock is true. But sometimes we need to test more. Sometimes we need to know not only if a value is greater than 32, but also if it's less than 100. Or, if an item is inStock and also onSale. Or that an item is on sale only on Tuesdays when the user is a VIP member. So, you see, these conditionals can get complex.

Let's step through a few to get a better idea of how they work.

Say we need to test that an item is inStock AND onSale. We could do that like this:

```
if (inStock == true) {
    if (onSale == true) {
        // sounds like a bargain!
        alert("buy buy buy!");
    }
}
```

First, see if the item is in stock...

And, if so, then see if it is on sale.

And if so, then take some action, like buy a few!

Notice this code is executed only if both conditionals are true!

We can simplify this code by combining these two conditionals together. Unlike in Simple Battleship, where we tested if guess < 0 OR guess > 6, here we want to know if inStock is true AND onSale is true. Let's see how to do that...

Here's our AND operator. With AND this combined conditional is true only if the first part AND the second part are true.

```
if (inStock == true  &&  onSale == true) {
          // sounds like a bargain!
          alert("buy buy buy!");
}
```

Not only is this code more concise, it's also more readable. Compare this code with the code on the previous page to see.

We don't have to stop there—we can use multiple boolean operators to combine conditionals in a variety of ways:

Now we're using both AND and OR in the same conditional expression. This one says: if an item is in stock AND it's either on sale OR the price is less than 60, then buy.

```
if (inStock == true  &&  (onSale == true || price < 60)) {
          // sounds like a bargain!
          alert("buy buy buy!");
}
```

Notice we're using parentheses to group the conditionals together so we get the result of the OR first, and then use that result to compute the result of the AND.

Sharpen your pencil

We've got a whole bunch of boolean expressions that need evaluating below. Fill in the blanks, and then check your answers at the end of the chapter before you go on.

```
let temp = 81;
let willRain = true;
let humid = (temp > 80 && willRain == true);
```

What's the value of **humid**? ______________

```
let guess = 6;
let isValid = (guess >= 0 && guess <= 6);
```

What's the value of **isValid**? ______________

```
let kB = 1287;
let tooBig = (kB > 1000);
let urgent = true;
let sendFile =
    (urgent == true || tooBig == false);
```

What's the value of **sendFile**? ______________

```
let keyPressed = "N";
let points = 142;
let level;
if (keyPressed == "Y" ||
     (points > 100 && points < 200)) {
   level = 2;
} else {
   level = 1;
}
```

What's the value of **level**? ______________

Solution on page 78

Exercise

Bob and Bill, both from accounting, are working on a new price checker application for their company's website. They've both written if/else statements using boolean expressions. Both are sure they've written the correct code. Which accountant is right? Should these accountants even be writing code? Check your answer at the end of the chapter before you go on.

Bob's solution

```
if (price < 200 || price > 600) {
   alert("Price is too low or too high! Don't buy the gadget.");
} else {
   alert("Price is right! Buy the gadget.");
}
```

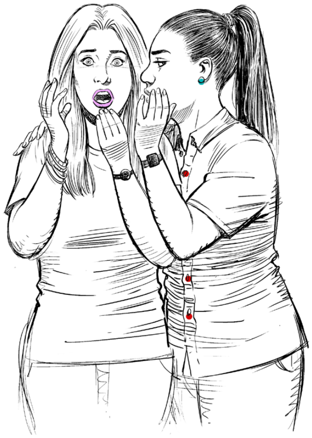
Who got it right? Bob or Bill?

Bill's solution

```
if (price >= 200 || price <= 600) {
   alert("Price is right! Buy the gadget.");
} else {
   alert("Price is too low or too high! Don't buy the gadget.");
}
```

Solution on page 79

Can we talk about your verbosity...



We don't know how to bring this up, but you've been a little verbose in specifying your conditionals. What do we mean? Take this condition, for instance:

We often compare our boolean variables to true or false to form our conditionals.

```
if  (inStock == true) {
    ...
}
```

And inStock is a variable that holds a boolean value of true or false.

As it turns out, that's a bit of overkill. The whole point of a conditional is that it evaluates to either true or false, but our boolean variable `inStock` already *is* one of those values. So, we don't need to compare the variable to anything; it can just stand on its own. That is, we can just write this instead:

```
if  (inStock) {
    ...
}
```

If we just use the boolean variable by itself, then if that variable is true, the conditional test is true, and the block is executed.

And if inStock is false, then the conditional test fails and the code block is skipped.

Now, while some might claim our original, verbose version was clearer in its intent, it's more common to see the more succinct version in practice. And you'll find the less verbose version easier to read as well.

Exercise

We've got two statements below that use the onSale and inStock variables in conditionals to figure out the value of the variable buyIt. Work through each possible value of inStock and onSale for both statements. Which version is the biggest spender?

```
let buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	true		
true	false		
false	true		
false	false		

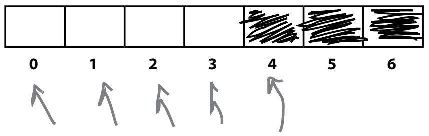
```
let buyIt = (inStock && onSale);
```

Solution on page 78

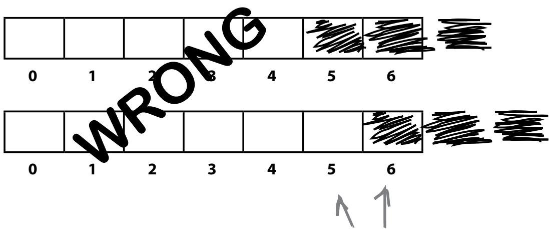
Finishing the Simple Battleship game

We still have one little matter to take care of, because right now we've hardcoded the location of the ship—no matter how many times you play the game, the ship is always at locations 3, 4, and 5. That actually works out well for testing, but we really need to randomly place the ship to make it a little more interesting for the user.

Let's step back and think about the right way to place a ship on the 1D grid of seven cells. We need a starting location that allows us to place it in three consecutive positions on the grid. That means we need a starting location from zero to four.



We can start in locations 0, 1, 2, 3, or 4 and still have room to place the ship in the next three positions...



...but, starting at position 5 or 6 won't work.

How to assign random locations

Once we have a starting location (between zero and four), we can simply use it and the following two locations to hold the ship:

```
let location1 = randomLoc;
let location2 = location1 + 1;
let location3 = location2 + 1;
```

Take the random location along with the next two consecutive locations.

Okay, but how do we generate a random number? That's where we turn to JavaScript and its built-in functions. More specifically, JavaScript comes with a bunch of built-in math-related functions, including a couple that can be used to generate random numbers. We're going to get deeper into built-in functions, and functions in general, a little later in the book. For now, we're just going to make use of these functions to get our job done.

The recipe for generating a random number

We're going to start with the `Math.random` function. Calling this function will give us back a random decimal number:

Our variable randomLoc. We want to assign a number from 0 to 4 to this variable.

Math.random is part of standard JavaScript and returns a random number.

```
let randomLoc = Math.random();
```

The only problem is it returns numbers like 0.128, 0.830, 0.9, and 0.42. These numbers are between 0 and 1 (not including exactly 1). So we need a way to use this to generate random numbers from 0–4.

What we need is an integer between 0 and 4—that is, 0, 1, 2, 3 or 4—not a decimal number, like 0.34. How can we get that? To start, we could multiply the number returned by `Math.random` by 5 to get a little closer; here's what we mean…

If we multiply the random number by 5, then we get a number between 0 and 5, but not including 5. Like 0.13983, 4.231, 2.3451, or 4.999.

```
let randomLoc = Math.random() * 5;
```

Remember, * means multiplication.

That's closer! Now all we need to do is clip off the end of the number to give us an integer number. To do that we can use another built-in Math function, `Math.floor`:

We can use Math.floor to round the number down to its nearest integer value.

```
let randomLoc = Math.floor(Math.random() * 5);
```

So, for instance, 0.13983 becomes 0, 2.34 becomes 2, and 4.999 becomes 4.

there are no Dumb Questions

Q: If we're trying to generate a number between 0 and 4, why did we multiply the value by 5? Why did we write `Math.floor(Math.random() * 5)`?

A: Good question. Math.random generates a number between 0 and 1, but not including 1. The maximum number you can get from Math.random is 0.999... When you multiply that number by 5, the highest number you'll get is 4.999… Math.floor always rounds a number down, so 1.2 becomes 1, but so does 1.9999. If we generate a number from 0 to 4.999… the rounded-down value will always be between 0 and 4. This is not the only way to do it, and in other languages it's often done differently, but this is how you'll see it done in most JavaScript code.

Q: So if I wanted a random number between 0 and 100 (including 100), I'd instead write `Math.floor(Math.random() * 101)`?

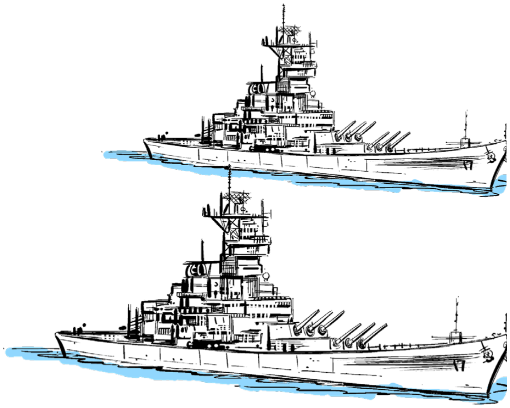
A: That's right! Multiplying by 101 and using Math.floor to round down ensures that your result will be at most 100.

Q: What are the parentheses for in Math.random()?

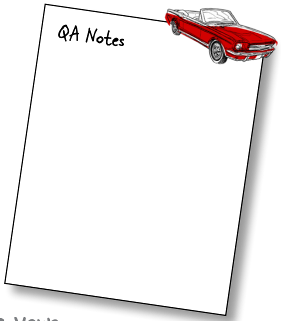
A: We use parentheses whenever we "call" a function. Sometimes we need to hand a value to a function, like we do when we use alert to display a message, and sometimes we don't, like when we use Math.random. But whenever you're calling a function (whether it's built-in or not), you'll need to use parentheses. Don't worry about this right now; we'll get into all these details in the next chapter.

Q: I can't get my Battleship game to work. I'm not seeing anything in my web page except the "Play battleship" heading. How can I figure out what I did wrong?

A: This is where using the console can come in handy. If you've made an error like forgetting a quote in a string, then JavaScript will typically complain about the syntax of your program not being right, and may even show you the line number where your error is. But some errors are more subtle. For instance, if you mistakenly write isSunk = false instead of isSunk == false, you won't see a JavaScript error, but your code won't behave as you expect it to. For this kind of error, try using console.log to display the values of your variables at various points in your code to see if you can track down the problem.



A little more QA

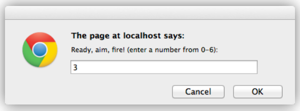


That's all we need. Put this code together (we've already done that below) and replace your existing location code with it. When you're finished, give it a few test runs to see how fast you can sink the enemy battleship.

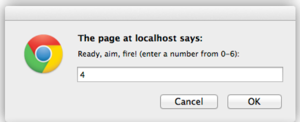
```
let randomLoc = Math.floor(Math.random() * 5);
let location1 = randomLoc;
let location2 = location1 + 1;
let location3 = location2 + 1;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      // the rest of your code goes here....
```

Go ahead and replace your location variable declarations with these new statements.

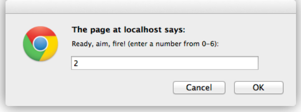
Here's one of our test sessions. The game's a little more interesting now that we've got random locations for the ship. But we still managed to get a pretty good score…


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
3

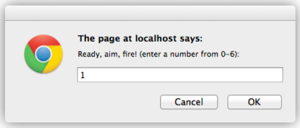
We get a hit on our first guess.


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
4

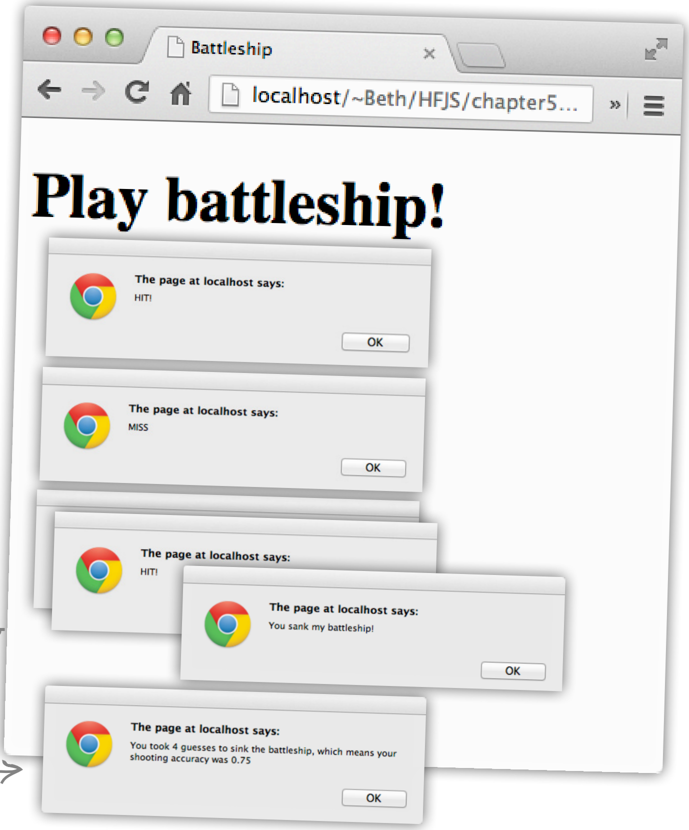
On our second guess, we miss.


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
2

But then we get two hits in a row.


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
1

On the last hit, we sink the battleship!



Play battleship!

The page at localhost says:
HIT!

The page at localhost says:
MISS

The page at localhost says:
HIT!

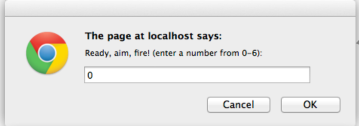
The page at localhost says:
You sank my battleship!

The page at localhost says:
You took 4 guesses to sink the battleship, which means your shooting accuracy was 0.75

Exercise

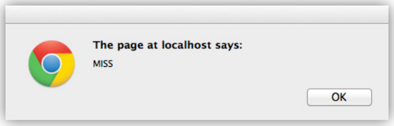
Wait a sec, we noticed something that might be a bug. Hint: when we enter 0, 1, 1, 1, things don't look right! Can you figure out what's happening?

Here are our guesses...

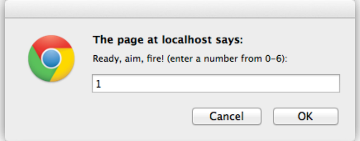


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
0

We miss on our first guess.

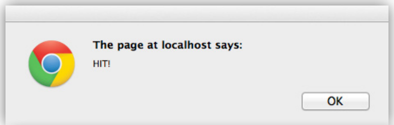


The page at localhost says:
MISS

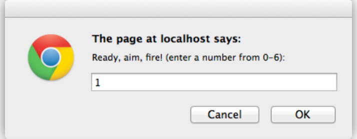


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
1

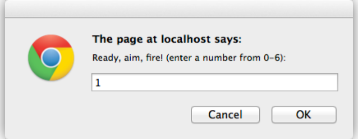
On our second guess we get a hit.



The page at localhost says:
HIT!

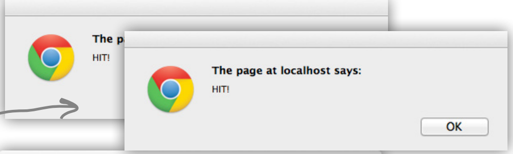


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
1



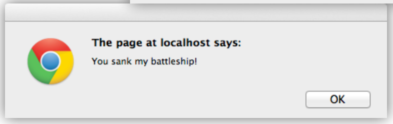
The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
1

And then we keep entering that same location, and keep getting hits!



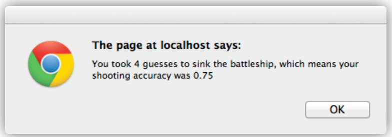
The p
HIT!

The page at localhost says:
HIT!



The page at localhost says:
You sank my battleship!

On the third hit, we see that we sank the battleship! **But something's wrong.** We shouldn't be able to sink it by hitting the same location three times.



The page at localhost says:
You took 4 guesses to sink the battleship, which means your shooting accuracy was 0.75

We entered 0, 1, 1, 1, and the ship is at 1, 2, 3.

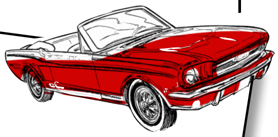
It's a cliff-hanger!

Will we ***find*** the bug?

Will we ***fix*** the bug?

Stay tuned for a much improved version of Battleship a little later in the book...

And in the meantime, see if you can come up with ideas for how you might fix the bug.



QA Notes

Found a bug! Entering the same number that is a hit on a ship results in sinking the ship, when it shouldn't.

Congrats on your first true JavaScript program, and a short word about reusing code

You've probably noticed that we've made use of a few *built-in functions* like `alert`, `prompt`, `console.log`, and `Math.random`. With very little effort, these functions have given you the ability to pop up dialog boxes, log output to the console, and generate random numbers, almost like magic. But these built-in functions are just packaged-up code that's already been written for you, and as you can see, their power is that you can use and reuse them just by making a call to them when you need them.

There's a lot to learn about functions, how to call them, what kinds of values you can pass them, and so on, and we're going to start getting into all that in the next chapter, where you'll learn to create your own functions.

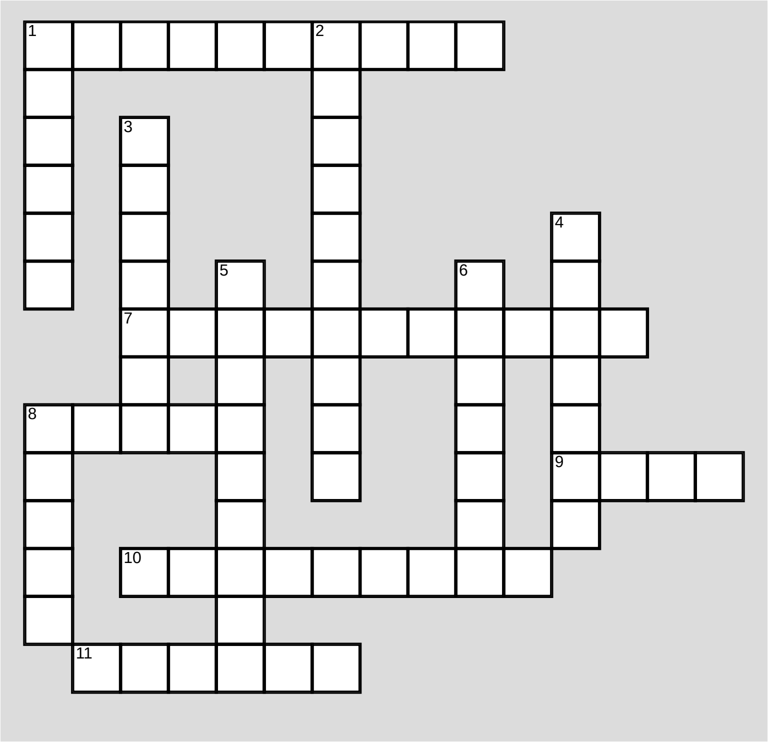
But before you get there, you've got the bullet points to review, a crossword puzzle to complete…oh, and a good night's sleep to let everything sink in.

Bullet Points

- You can use a flowchart to outline the logic of a JavaScript program, showing decision points and actions.
- Before you begin writing a program, it's a good idea to sketch out what your program needs to do with pseudocode.
- **Pseudocode** is an approximation of what your real code should do.
- There are two kinds of boolean operators: comparison operators and logical operators. When used in an expression, boolean operators result in a true or false value.
- **Comparison** operators compare two values and result in true or false. For example, we can use the boolean comparison operator < ("less than") like this: 3 < 6. This expression results in true.
- **Logical** operators combine two boolean values. For example, true || false results in true; true && false results in false.
- You can generate a random number between 0 and 1 (including 0, but not including 1) using the **Math.random** function.
- The **Math.floor** function rounds down a decimal number to the nearest integer.
- Make sure you use Math with an uppercase M, and not m, when using Math.random and Math.floor.
- The JavaScript function **prompt** shows a dialog with a message and a space for the user to enter a value.
- In this chapter, we used prompt to get input from the user and **alert** to display the results of the Battleship game in the browser.

JavaScript cross

How does a crossword puzzle help you learn JavaScript? The mental twists and turns burn the JavaScript right into your brain!



ACROSS

1. This helps you think about how a program is going to work.

7. Both while and if statements use _______ tests.

8. Boolean operators always result in true or _______.

9. To get a true value from an AND operator (&&), both parts of the conditional must be _______.

10. JavaScript has many built-in _______, like alert and prompt.

11. To randomly choose a position for a ship, use Math._______.

Solution on page 82

DOWN

1. To get input from a user, you can use the _______ function.

2. == is a _______ operator you can use to test to see if two values are the same.

3. OR (||) and AND (&&) are _______ operators.

4. If you're good at testing programs, you might want to become a _______ assurance specialist.

5. If you don't initialize a variable, the value is _______.

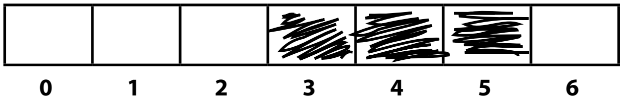
6. We keep track of whether a ship is sunk or not with a _______ variable.

8. To get a false value from an OR operator (||), both parts of the conditional must be _______.

Sharpen your pencil Solution

From page 52

Let's say our virtual grid looks like this:



And we've represented the ship's location using our location variables, like this:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Use the following sequence as your test user input:

```
1, 4, 2, 3, 5
```

Now, using the pseudocode on page 51, trace through each step of the code and see how this works given the user input. Put your notes below. Here's our solution.

location1	location2	location 3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false
3	4	5	4	2	1	false
3	4	5	2	3	1	false
3	4	5	3	4	2	false
3	4	5	5	5	3	true

Exercise Solution

From page 69

We've got two statements below that use the onSale and inStock variables in conditionals to figure out the value of the variable buyIt. Work through each possible value of inStock and onSale for both statements. Which version is the biggest spender? The OR (||) operator!

```
let buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

```
let buyIt = (inStock && onSale);
```

Sharpen your pencil Solution

From page 67

We've got a whole bunch of boolean expressions that need evaluating below. Fill in the blanks. Here's our solution.

```
let temp = 81;
let willRain = true;
let humid = (temp > 80 && willRain == true);
```

What's the value of **humid**? true

```
let guess = 6;
let isValid = (guess >= 0 && guess <= 6);
```

What's the value of **isValid**? true

```
let kB = 1287;
let tooBig = (kB > 1000);
let urgent = true;
let sendFile =
     (urgent == true || tooBig == false);
```

What's the value of **sendFile**? true

```
let keyPressed = "N";
let points = 142;
let level;
if (keyPressed == "Y" ||
     (points > 100 && points < 200)) {
    level = 2;
} else {
    level = 1;
}
```

What's the value of **level**? 2

Exercise Solution

From page 68

Bob and Bill, both from accounting, are working on a new price checker application for their company's website. They've both written if/else statements using boolean expressions. Both are sure they've written the correct code. Which accountant is right? Should these accountants even be writing code? Here's our solution.

Bob's solution

```
if (price < 200 || price > 600) {
   alert("Price is too low or too high! Don't buy the gadget.");
} else {
   alert("Price is right! Buy the gadget.");
}
```

Bill's solution

```
if (price >= 200 || price <= 600) {
   alert("Price is right! Buy the gadget.");
} else {
   alert("Price is too low or too high! Don't buy the gadget.");
}
```

Bob's the better coder (and possibly a better accountant, too). Bob's solution works, but Bill's doesn't. To see why, let's try three different prices (too low, too high, and just right) with Bob's and Bill's conditionals and see what results we get.

price	Bob's	Bill's
100	true alert: Don't buy!	true alert: Buy!
700	true alert: Don't buy!	true alert: Buy!
400	false alert: Buy!	true alert: Buy!

If price is 100, then 100 is less than 200, so Bob's conditional is true (remember, with OR, you only need one of the expressions to be true for the whole thing to be true), and we alert NOT to buy.

But Bill's conditional is also true, because price is <= 600. So the result of the entire expression is true, and we alert the user to buy, even though the price is too low.

Turns out Bill's conditional is always true, no matter what the price is, so his code tells us to Buy! every time. Bill should stick with accounting.

Exercise Solution

From page 63

Remember we said pseudocode often isn't perfect? Well, we actually left something out of our original pseudocode: we're not telling the user if her guess is a HIT or a MISS. Can you insert the missing pieces of code in the proper place to correct this? Here's our solution.

```
// Variable declarations go here


while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {

            alert("HIT!");
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        }
        else {
            alert("MISS");
    }
        }
}
let stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Sharpen your pencil Solution

From page 60

What do you think of this first attempt to write the code to detect when a ship is hit? Does it look more complex than it needs to be, or are we repeating code in a way that seems a bit, well, redundant? Could we simplify it? Using what you know of the || operator (that is, the boolean OR operator), can you simplify this code? Here's our solution.

```
if (guess == location1) {
     hits = hits + 1;
} else  if (guess == location2) {
     hits = hits + 1;
} else if (guess == location3) {
     hits = hits + 1;
}
```

We're using the same code over and over here.

If we ever have to change how hits are updated, we've got three places to change our code. Changes like this are often a source of bugs and issues in code.

Not only that, this code is just way more complex than it needs to be. It's harder to read than it should be, and it took a lot more thought and typing than needed.

But, with the boolean OR operator we can combine the tests so that if guess matches any of location1, location2, or location3, the if conditional will be true and the hits variable will be updated.

```
if (guess == location1 || guess == location2 || guess == location3) {
     hits = hits + 1;
}
```

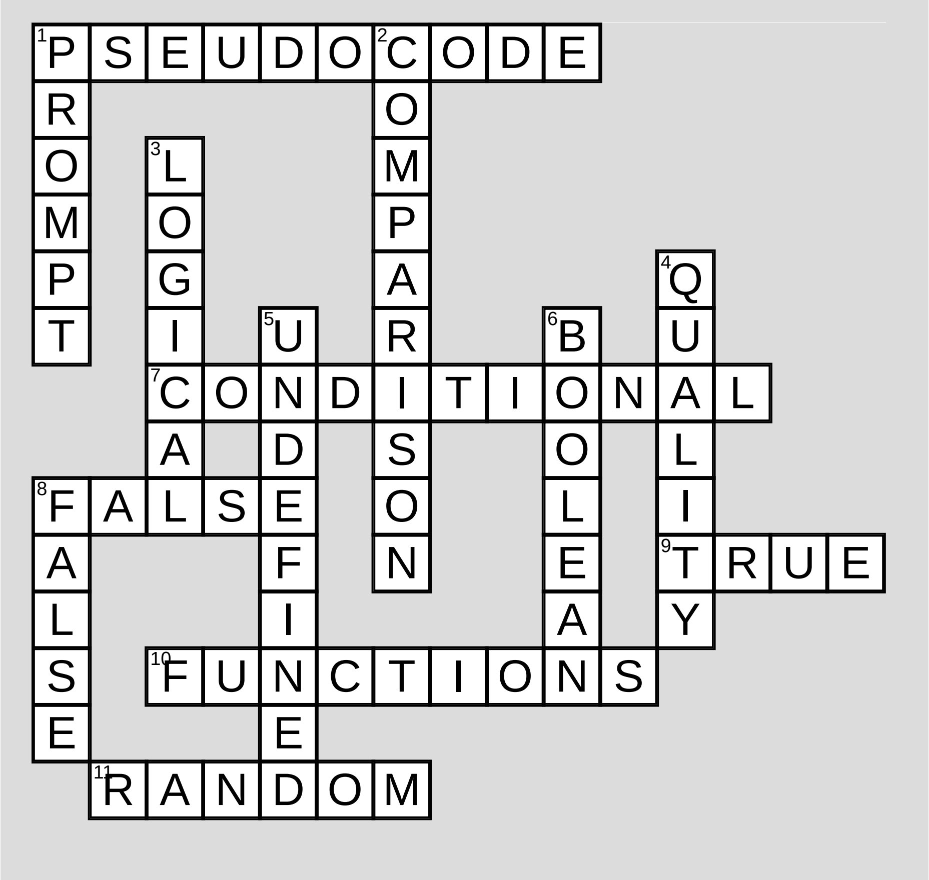
Isn't that much easier on the eye? Not to mention easier to understand.

And if we ever have to change how hits is updated, well, then we only have one place to do it, which is much less error-prone.

JavaScript cross Solution

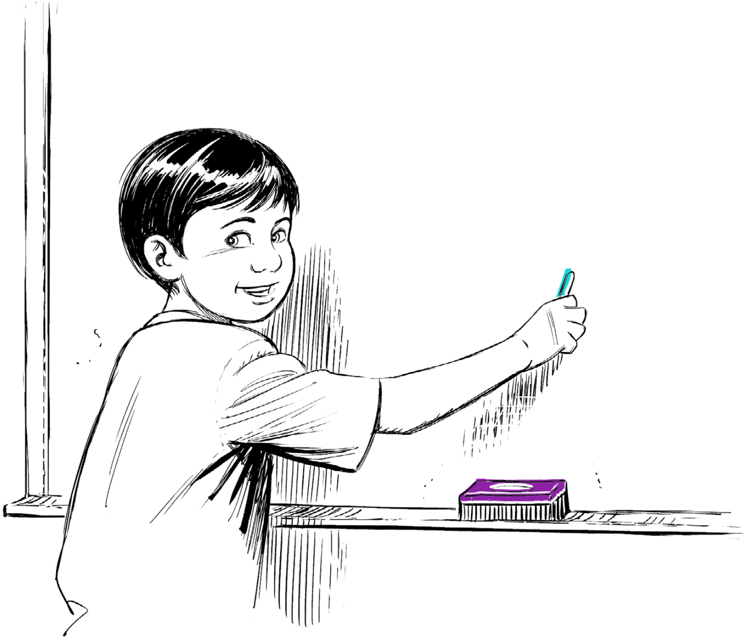
From page 76

How does a crossword puzzle help you learn JavaScript? The mental twists and turns burn the JavaScript right into your brain! Here's our solution.


1 PSEUDOCODE
1 PROMPT
2 COMPARISON
3 LOGICAL
4 QUALITY
5 UNDEFINED
6 BOOLEAN
7 CONDITIONAL
8 FALSE
8 FALSE
9 TRUE
10 FUNCTIONS
11 RANDOM

3 introducing functions

Getting functional



Get ready for your first superpower. You've got some programming under your belt; now it's time to really move things along with **functions**. Functions give you the power to write code that can be applied to all sorts of different circumstances, code that can be **reused** over and over, code that is much more **manageable,** code that can be **abstracted** away and given a simple name so you can forget all the complexity and get on with the important stuff. You're going to find not only that functions are your gateway from scripter to programmer, but that they're the key to the JavaScript programming style. In this chapter, we're going to start with the basics—the mechanics, the ins and outs of how functions really work—and then you'll keep honing your function skills throughout the rest of the book. So, let's get a good foundation started, *now*.

More on this as we progress through the book.

Sharpen your pencil

Do a little analysis of the code below. How does it look? Choose among the options at the bottom of the page, or write in your own analysis.

```
let dogName = "rover";
let dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

☐ A. The code seems very redundant.

☐ B. If we want to change the display of the output, or add another weight for dogs, this is going to require a lot of reworking.

☐ C. Looks tedious to type in!

☐ D. Not the most readable code I've ever seen.

☐ E. ______________________________
____

→ Solution on page 122

What's wrong with the code, anyway?

We just looked at some code that got used *over and over*. What's wrong with that? Well, at face value, nothing. After all, it works, right? Let's take a closer look at the code in question:

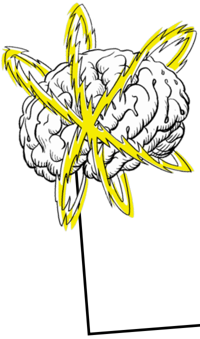
```
let dogName = "rover";
let dogWeight = 23;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
 ...
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
```

What we're doing here is comparing the dog's weight to 20, and if it's greater than 20, we're outputting a big WOOF WOOF. If it's less than 20, we're outputting a smaller woof woof.

And this code is...d'oh! It's doing EXACTLY the same thing. And so on, many times over in the rest of the code.

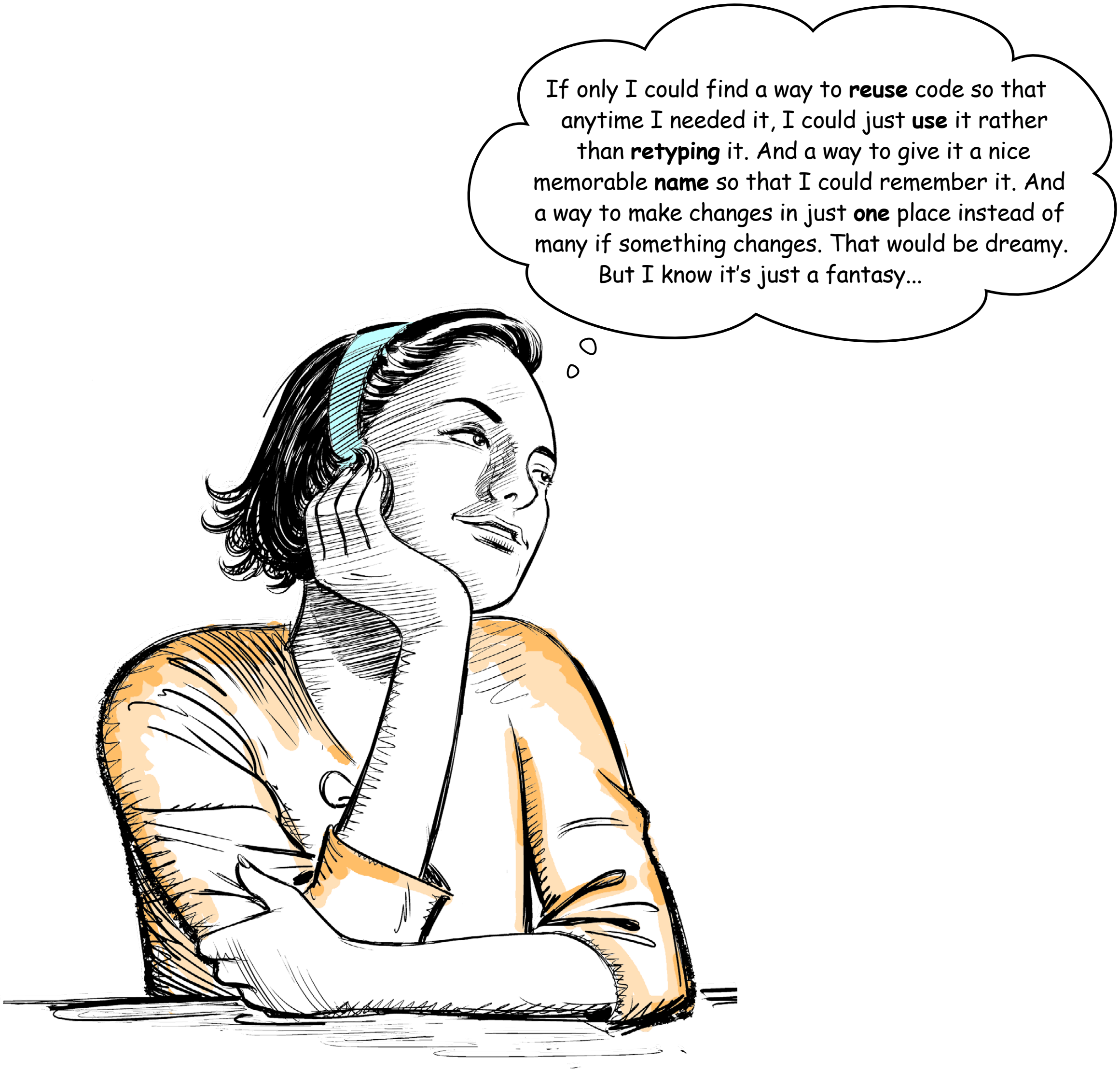
```
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

Sure, this code looks innocent enough, but it's tedious to write, a pain to read, and will be problematic if it needs to change over time. That last point will ring true more and more as you gain experience in programming. All code changes over time, and the code above is a nightmare waiting to happen because the same logic is repeated over and over, so if you need to change that logic, you'll have to change it in multiple places. And the bigger the program gets, the more changes you'll have to make, leading to more opportunities for mistakes. What we really want is a way to take redundant code like this and put it in one place where it can be easily reused whenever we need it.



Brain Power

How can we improve this code? Take a few minutes to think of a few possibilities. Does JavaScript have something that could help?


If only I could find a way to **reuse** code so that anytime I needed it, I could just **use** it rather than **retyping** it. And a way to give it a nice memorable **name** so that I could remember it. And a way to make changes in just **one** place instead of many if something changes. That would be dreamy. But I know it's just a fantasy...

By the way, did we happen to mention FUNCTIONS?

Meet *functions*. JavaScript functions allow you to take a bit of code, give it a name, and then refer to it over and over whenever you need it. That sounds like just the medicine we're looking for.

Say you're writing some code that does a lot of "barking" (like in the previous exercise). If your code is dealing with a big dog, then the bark is a big "WOOF WOOF." And if it's a small dog, the bark is a tiny "woof woof." You're going to need to use this barking functionality many times in your code. Let's write a `bark` function you can use over and over:

The function keyword begins a function definition.

Then we give the function a name, like bark.

And we're going to hand it two things when we get around to using it: a dog name and a dog weight.

We call these the parameters of the function. We put these in parentheses after the function name.

```
function bark(name, weight) {


}
```

Next, we're going to write some code that gets executed when we use the function.

We'll call this the body of the function. It's everything inside the { and the }.

Now we need to write the code for the function. Our code will check the weight and output the appropriate-sized bark:

First we need to check the weight, and...

Notice the variable names used in the code match the parameters of the function.

```
function bark(name, weight) {
    if (weight > 20) {
        console.log(name + " says WOOF WOOF");
    } else {
        console.log(name + " says woof woof");
    }
}
```

...then output the dog's name with WOOF WOOF or woof woof.

Now you have a function you can use in your code. We'll see how that works next...

Okay, but how does it actually work?

First, let's rework our code using the new function, `bark`:

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}

bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

Ahh, this is nice, all the logic of the code is here in one place.

Now all that code becomes just a few calls to the bark function, passing it each dog's name and weight.

Now that's simple!

Wow, that's a lot less code—and it's so much more readable to your coworker who needs to go into your code and make a quick change. We've also got all the logic in one convenient location.

But how exactly does it all come together and actually work? Let's go through it step by step.

First we have the function.

So, we've got the `bark` function right at the top of the code. The browser reads this code, sees it's a function, and then takes a look at the statements in the body. The browser knows it isn't executing the function statements now; it'll wait until the function is called from somewhere else in the code.

Notice too that the function is *parameterized*, meaning it takes a dog's name and weight when it is called. That allows you to call this function for as many different dogs as you like. Each time you do, the logic applies to the name and weight you pass to the function call.

Again, these are parameters; they are assigned values when the function is called.

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

And everything inside the function is the body of the function.

Now let's call the function.

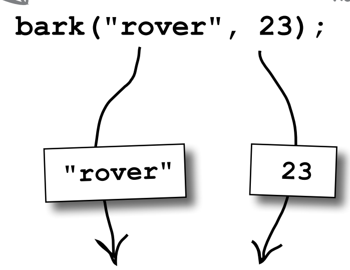
"Invoking a function" is just a fancy way of saying "calling a function." Feel free to mix and match, especially when your new boss is around.

To call, or *invoke*, a function, just use its name, followed by an open parenthesis, then any values you need to pass it (separated by commas), and finally a closing parenthesis. The values in the parentheses are *arguments*. For the `bark` function, we need two arguments: the dog's name and the dog's weight.

Here's how the call works.

Our function name.

Here we're passing two arguments, the name and the weight.



```
bark("rover", 23);
```

```
function bark(name, weight) {
    if (weight > 20) {
        console.log(name + " says WOOF WOOF");
    } else {
        console.log(name + " says woof woof");
    }
}
```

When we call the bark function, the arguments are assigned to the parameter names.

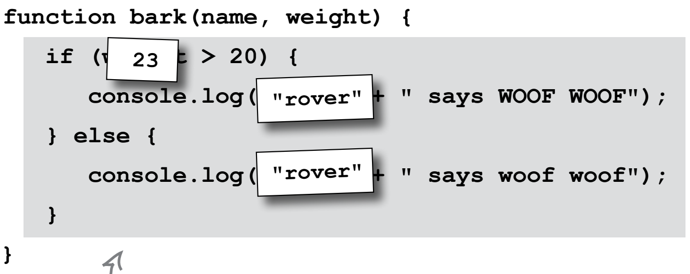
And any time the parameters appear in the function, the values we passed in are used.

After you call the function, the body of the function does all the work.

Once we know the value for each parameter—like `name` is "rover" and `weight` is 23—we're ready to execute the function body.

Statements in the function body are executed from top to bottom, just like all the other code you've been writing. The only difference is that the parameter names `name` and `weight` have been assigned the values of the arguments you passed into the function.

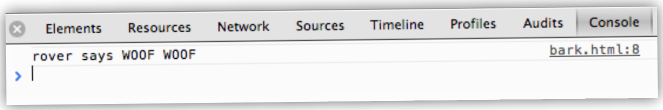
After we've assigned the argument values to the parameter names, we execute the statements in the body.



```
function bark(name, weight) {
    if (23 > 20) {
        console.log("rover" + " says WOOF WOOF");
    } else {
        console.log("rover" + " says woof woof");
    }
}
```

The parameters act like variables in the body, which have been assigned the values of the arguments you passed in.

Use your browser's Developer Tools to access the console so you can see the output of bark.



```
rover says WOOF WOOF                                      bark.html:8
> |
```

And when it's done... The logic of the body has been carried out (and, in this example, you'll see that Rover, being 23 pounds, says "WOOF WOOF"), and the function is done. After the function completes, control is returned to the statement following our call to `bark`.

"rover says WOOF WOOF"

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

We just did this...

...so do this next.

When the function completes, the browser starts executing the next line of code after where we called the function.

Here, we're calling the function again, with different arguments, so the process starts all over again!

Sharpen your pencil

We've got some more calls to bark below. Next to each call, write what you think the output should be, or if you think the code will cause an error. Check your answers at the end of the chapter before you go on.

`bark("juno", 20);` ____________________

`bark("scottie", -1);` ____________________

`bark("dino", 0, 0);` ____________________

`bark("fido", "20");` ____________________

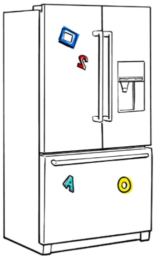
`bark("lady", 10);` ____________________

`bark("bruno", 21);` ____________________

Write what you think the console log will display here.

Hmm, any ideas what these do?

Solution on page 123



Code Magnets

This working JavaScript code is all scrambled up on the fridge. Can you reorder the code snippets to make a working program that produces the output listed below? Note that there may be some extra code on the fridge, so you may not use all the magnets. Check your answer at the end of the chapter.

```
,
```

```
,
```

```
,
```

```
}
```

```
}
```

```
}
```

```
(
```

```
function
```

```
)
```

```
{
```

```
whatShallIWear(80);
```

```
else {
    console.log("Wear a t-shirt");
}
```

```
whatShallIWear
```

```
else if (temp < 70) {
     console.log("Wear a sweater");
}
```

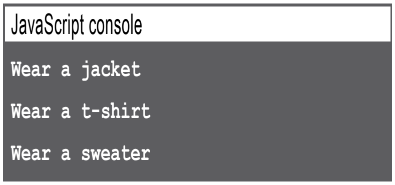
```
temperature
```

```
if (temp < 60) {
     console.log("Wear a jacket");
}
```

```
temp
```

```
whatShallIWear(60);
```

```
whatShallIWear(50);
```



JavaScript console

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

We're using this to represent a generic console.

Solution on page 124

The Function Exposed

This week's interview: The intimate side of functions...

Head First: Welcome, Function! We're looking forward to digging in and finding out what you're all about.

Function: Glad to be here.

Head First: Now, we've noticed many JavaScript newbies tend to ignore you. They just get in and write their code, line by line, top to bottom, no functions at all. Are you really needed?

Function: Those newbies are missing out. That's unfortunate, because I'm powerful. Think about me like this: I give you a way to write code once and then reuse it over and over.

Head First: Well, excuse me for saying this, but if you're just giving them the ability to do the same thing, over and over...that's a little boring isn't it?

Function: No no, functions are parameterized—in other words, each time you use the function, you pass it arguments so it can compute something that's relevant to what you need.

Head First: Err, example?

Function: Let's say you need to show a user how much the items in his shopping cart are going to cost, so you write a function named `computeShoppingCartTotal`. I can compute the amount for any shopping cart you give me.

Head First: If you're so great, why aren't more new coders using you?

Function: That's not even a true statement; they use me all the time! `alert`, `console.log`, `prompt`, `Math.random`...it's hard to write anything meaningful without using functions. It's not so much that new users don't use functions, they just aren't defining *their own* functions.

Head First: Well, right, `alert` and `prompt`, those make sense, but take `Math.random`—that doesn't look quite like a function.

Function: `Math.random` is a function, but it happens to be attached to another powerful thing new coders don't make a lot of use of: *objects*.

Head First: Oh yes, objects. I believe our readers are learning about those in a later chapter.

Function: Fair enough, I'll save my breath on that one for later.

Head First: Now, this argument/parameter stuff all seems a little confusing.

Function: Think about it like this: each parameter acts like a variable throughout the body of the function. When you call the function, each value you pass in is assigned to a corresponding parameter.

Head First: And arguments are what?

Function: Oh, that's just another name for the values you pass into a function…they're the arguments of the function call.

Head First: Well, you don't seem all that great; I mean, okay, you allow me to reuse code, and you have this way of passing values into parameters. Is that it? I don't get the mystery around you.

Function: Oh, that's just the basics. There's so much more: I can return values, I can masquerade around your code anonymously, I can do a neat trick called closures, and I have an *intimate* relationship with objects.

Head First: Ohhhhh REALLY?! Can we get an exclusive on that relationship for our next interview?

Function: We'll talk…

What can you pass to a function?

When you call a function, you pass it arguments, and those arguments then get matched up with the parameters in the function definition. You can pass any JavaScript value as an argument, like a string, a boolean, or a number:

Pass any JavaScript value as an argument.

```
saveMyProfile("krissy", 1991, 3.81, false);

function saveMyProfile(name, age, GPA, newuser) {
    if (age <= 17) {
        // code for handling a child
    }
   // rest of code for this function here
}
```

Each argument is passed to its corresponding parameter in the function.

And each parameter acts as a variable within the function.

You can also pass variables as arguments, and that's often the more common case. Here's the same function call using variables:

```
let student = "krissy";
let year = 2024;
let GPA = 381/100;
let status = "existing user";
let isNewUser = (status == "new user");
saveMyProfile(student, year, GPA, isNewUser);
```

Now, each of the values we're passing is stored in a variable. When we call the function, the variable's values are passed as the arguments.

So, in this case we're passing the value in the variable student, "krissy", as the argument to the name parameter.

And we're also using variables for these other arguments.

And you can even use expressions as arguments:

```
let student = "krissy";
let status = "existing user";
let year = 2024;

saveMyProfile(student, year, 381/100, status == "new user");
```

Yes, even these expressions will work as arguments!

In each case we first evaluate the expression to a value, and then that value is passed to the function.

We can evaluate a numeric expression...

...or a boolean expression, like this one that results in passing false to the function.

I'm still not sure I get the difference between a parameter and an argument—are they just two names for the same thing?

No, they're different.

When you define a function, you can *define* it with zero or more *parameters*:

Here we're defining three parameters: degrees, mode, and duration.

```
function cook(degrees, mode, duration) {
    // your code here
}
```

When you call a function, you *call* it with *arguments*:

```
cook(425.0, "bake", 45);
```

These are arguments. There are three arguments: a floating-point number, a string, and an integer.

```
cook(350.0, "broil", 10);
```

So you'll only define your parameters once, but you'll probably call your function with many different arguments.

Brain Power

What does this code output? Are you sure?

```
function doIt(param) {
    param = 2;
}
let test = 1;
doIt(test);
console.log(test);
```

Sharpen your pencil

Below you'll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify all the variables, functions, arguments, and parameters. Write the names of each in the appropriate boxes on the right. Check your answers at the end of the chapter before you go on.

```
function dogYears(dogName, age) {
    let years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
const myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
let guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    const defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
let pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables (and constants)

Functions

Built-in functions

Arguments

Parameters

Solution on page 125

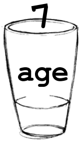
JavaScript is pass-by-value

(that means pass-by-copy)

It's important to understand how JavaScript passes arguments. JavaScript passes arguments to a function using *pass-by-value*. What that means is that each argument is *copied* into the parameter variable. Let's look at a simple example to see how this works:

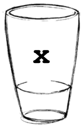
1. Let's declare a variable age, and initialize it to the value 7.

```
let age = 7;
```



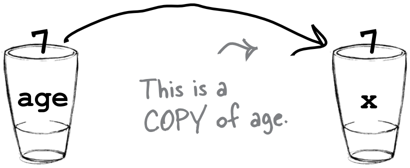
2. Now let's declare a function addOne, with a parameter named x, that adds 1 to the value of x.

```
function addOne(x) {
    x = x + 1;
}
```



3. Now let's call the function addOne and pass it the variable age as the argument. The value in age is copied into the parameter x.

```
addOne(age);
```

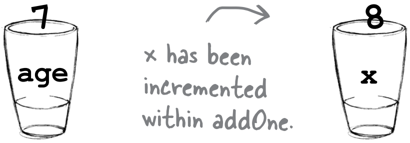


This is a COPY of age.

4. Now the value of x is incremented by one. But remember x is a copy, so only x is incremented, not age.

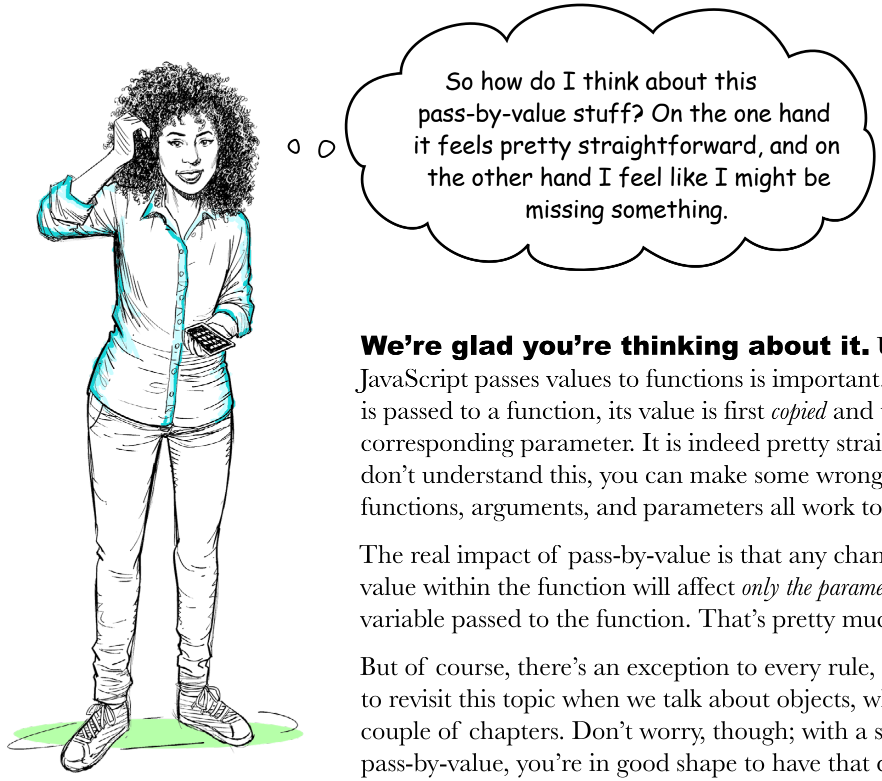
We're incrementing x.

```
function addOne(x) {
    x = x + 1;
}
```



x has been incremented within addOne.

age doesn't change even if x does.


So how do I think about this pass-by-value stuff? On the one hand it feels pretty straightforward, and on the other hand I feel like I might be missing something.

We're glad you're thinking about it. Understanding how JavaScript passes values to functions is important. When an argument is passed to a function, its value is first *copied* and then assigned to the corresponding parameter. It is indeed pretty straightforward, but if you don't understand this, you can make some wrong assumptions about how functions, arguments, and parameters all work together.

The real impact of pass-by-value is that any changes to a parameter's value within the function will affect *only the parameter,* not the original variable passed to the function. That's pretty much it.

But of course, there's an exception to every rule, and we're going to have to revisit this topic when we talk about objects, which we'll get to in a couple of chapters. Don't worry, though; with a solid understanding of pass-by-value, you're in good shape to have that discussion.

And for now, just remember that because of pass-by-value, *whatever happens to a parameter in the function, stays in the function.* Kinda like Vegas.

Brain Power Revisited

Remember this Brain Power from page 94? Do you think about it differently now, knowing about pass-by-value? Or did you guess correctly the first time?

```
function doIt(param) {
    param = 2;
}
let test = 1;
doIt(test);
console.log(test);
```

Remember this Brain Power?

```
function doIt(param) {
    param = 2;
}
const test = 1;
doIt(test);
console.log(test);
```

What about this one? It uses const.

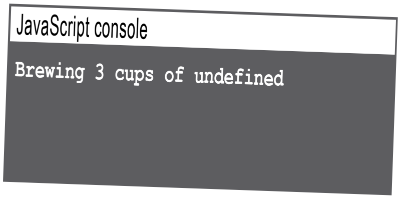
Weird Functions

So far you've seen the normal way to use functions, but what happens when you experiment a little by, say, passing too many arguments to a function? Or not enough? Sounds dangerous. Let's see what happens.

EXPERIMENT #1: What happens when we don't pass enough arguments?

Sounds dicey, but all that really happens is each parameter that doesn't have a matching argument is set to undefined. Here's an example:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3);
```



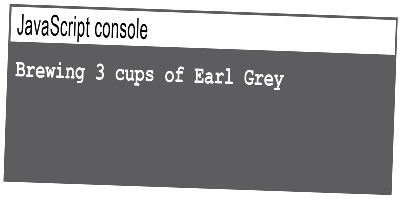
```
Brewing 3 cups of undefined
```

Notice that the value of the parameter tea is undefined because we didn't pass in a value.

EXPERIMENT #2: What happens when we pass too many arguments?

Ah, in this case JavaScript just ignores the extra arguments. Here's an example:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3, "Earl Grey", "hey ma!", 42);
```



```
Brewing 3 cups of Earl Grey
```

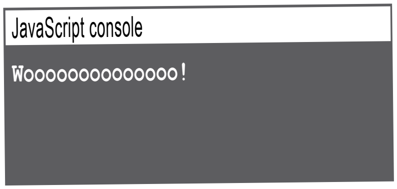
Works fine, the function ignores the extras.

There's actually a way to get at the extra arguments, but we won't worry about that just now...

EXPERIMENT #3: What happens when we have NO parameters?

No worries, many functions have no parameters!

```
function barkAtTheMoon() {
  console.log("Woooooooooooooo!");
}
barkAtTheMoon();
```



```
Woooooooooooooo!
```

Functions can return things too

You know how to communicate with your functions in one direction; that is, you know how to pass arguments *to functions*. But what about the other way? Can a function communicate back? Let's check out the `return` statement:

```
function bake(degrees) {
    let message;

    if (degrees > 500) {
        message = "I'm not a nuclear reactor!";
    } else if (degrees < 100) {
        message = "I'm not a refrigerator!";
    } else {
        message = "That's a very comfortable temperature for me.";
        setMode("bake");
        setTemp(degrees);
    }
    return message;
}

let status = bake(350);
```

Here we've got a new bake function that takes the temperature in degrees for the oven.

It then sets a variable to a string that depends on the temperature requested in the degrees parameter.

And presumably some real work is getting done here, but we won't worry about those details for now...

What we care about is that a return statement is returning the message as the result of this function.

Now, when the function is called and returns, the string that is returned as a result will be assigned to the status variable.

And in this case, if the status variable was printed, it would hold the string "That's a very comfortable temperature for me." Work through the code and make sure you see why!

Tracing through a function with a return statement

Now that you know how arguments and parameters work, and how you can return a value from a function, let's trace through a function call from start to finish to see what happens at every step along the way. Be sure to follow the steps in order.

① First, we declare a constant radius and initialize it to 5.2.

② Next, we call the calculateArea function and pass the radius variable as the argument.

③ The argument is sent to the parameter r, and the calculateArea function begins executing with r containing the value 5.2.

④ The body of the function executes, starting with declaring a variable, area. We then test to see if the parameter r has a value <= 0.

⑤ If r <= 0, then we return 0 from the function and the function stops executing. But we passed in 5.2, so this line does NOT execute.

⑥ We execute the else clause instead.

⑦ We compute the area of the circle using the value 5.2 in the parameter r.

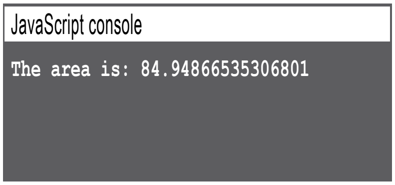
⑧ We return the value of area from the function. This stops the execution of the function and returns the value.

⑨ The value returned from the function is stored in the variable theArea.

⑩ Execution continues on the next line.

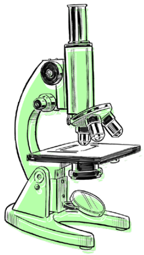
```
③ function calculateArea(r) {
       let area;
④      if (r <= 0) {
    ⑤     return 0;
⑥      } else {
⑦         area = Math.PI * r * r;
⑧         return area;
       }
  }
① const radius = 5.2;
                    ②
⑨ let theArea = calculateArea(radius);

⑩ console.log("The area is: " + theArea);
```

Output here!



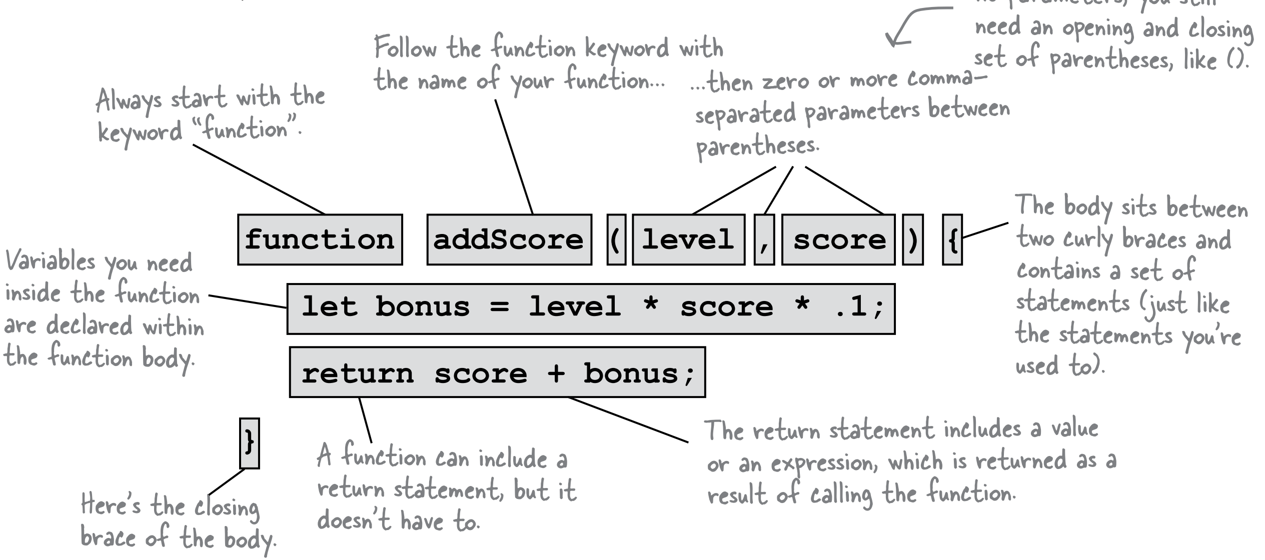
```
The area is: 84.94866535306801
```

Developers often call this "tracing the flow of execution," or just "tracing." As you can see, the flow can jump around when you're calling functions and returning values. Just take it slow, one step at a time.



Anatomy of a Function

Now that you know how to define and call a function, let's make sure you've got the syntax down cold. Here are all the parts of a function's anatomy:



```
function addScore(level, score) {
    let bonus = level * score * .1;
    return score + bonus;
}
```

Always start with the keyword "function".

Follow the function keyword with the name of your function...

...then zero or more comma-separated parameters between parentheses.

Even if your function has no parameters, you still need an opening and closing set of parentheses, like ().

The body sits between two curly braces and contains a set of statements (just like the statements you're used to).

Variables you need inside the function are declared within the function body.

Here's the closing brace of the body.

A function can include a return statement, but it doesn't have to.

The return statement includes a value or an expression, which is returned as a result of calling the function.

there are no Dumb Questions

Q: What happens if I mix up the order of my arguments, so that I'm passing the wrong arguments into the parameters?

A: All bets are off; in fact, we'd guess you're pretty much guaranteed either an error at runtime or an incorrect result. Always take a careful look at a function's parameters, so you know what arguments the function expects to be passed in.

Q: Can I pass a constant to a function?

A: You sure can. Its value will be passed to the function, just like with a variable declared with let.

Q: Why don't the parameter names have let or const in front of them? A parameter is a new variable, right?

A: Effectively, yes. The function does all the work of instantiating the variables for you, so you don't need to supply the let or const keyword in front of your parameter names.

Q: What are the rules for function names?

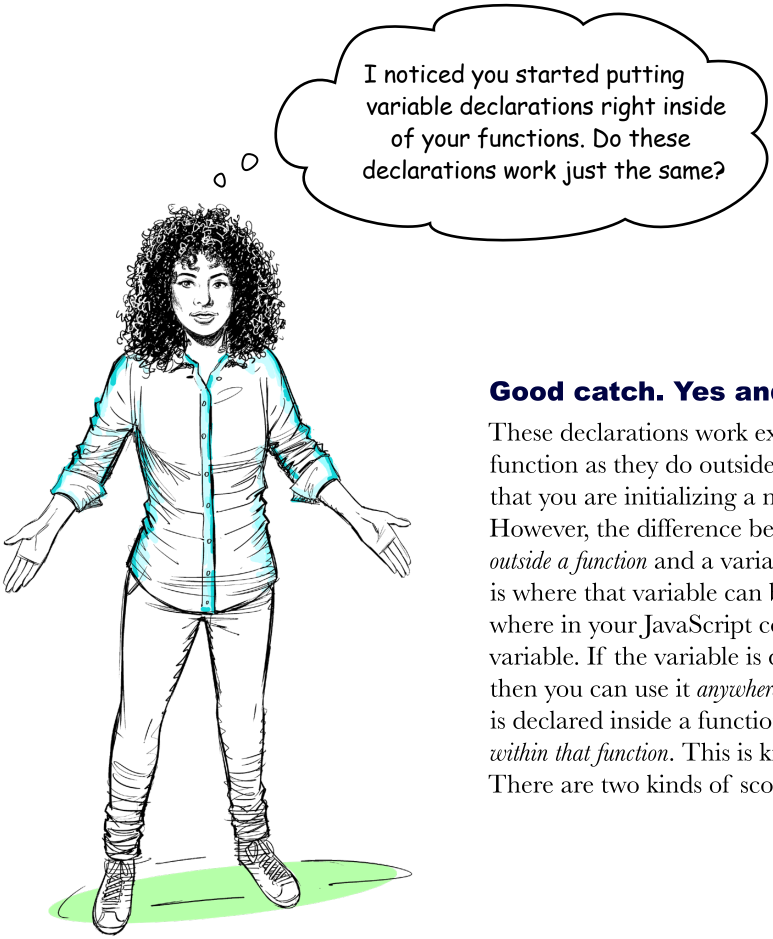
A: The rules for naming a function are the same as the rules for naming a variable. Just like with variables, you'll want to use names that make sense to you when you read them and provide some indication of what the function does, and you can use camel case (e.g., camelCase) to combine words in function names, just like with variables.

Q: What happens if I use the same name for an argument variable as the parameter? Like if I use the name x for both?

A: Even if your argument and parameter have the same name, like x, the parameter x gets a copy of the argument x, so they are two different variables. Changing the value of the parameter x does not change the value of the argument x.

Q: What does a function return if it doesn't have a return statement?

A: A function without a return statement returns `undefined`.

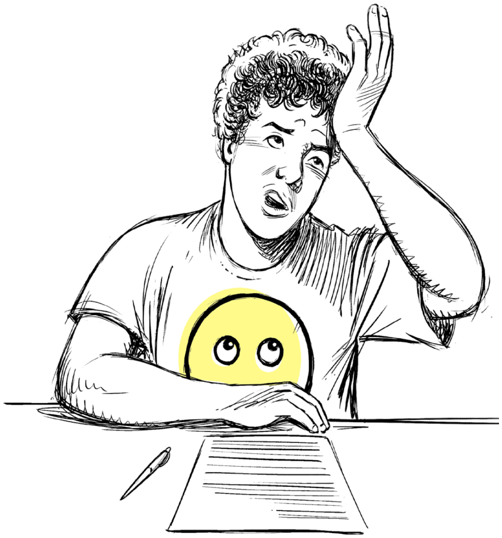

I noticed you started putting variable declarations right inside of your functions. Do these declarations work just the same?

Good catch. Yes and no.

These declarations work exactly the same within a function as they do outside a function, in the sense that you are initializing a new variable to a value. However, the difference between a variable declared *outside a function* and a variable declared *inside a function* is where that variable can be used—in other words, where in your JavaScript code you can reference the variable. If the variable is declared outside a function, then you can use it *anywhere* in your code. If a variable is declared inside a function, then you can use it only *within that function*. This is known as a variable's *scope*. There are two kinds of scope: global and local.

Global and local variables

Know the difference or risk humiliation



You already know that you can declare a variable by using the `let` or `const` keyword and a name anywhere in your script:

```
let avatar;
const levelThreshold = 1000;
```

These are global variables and constants; they're accessible everywhere in your JavaScript code.

And you've seen that you can also declare variables inside a function:

```
function getScore(points) {
    let score;
    let i = 0;
    while (i < levelThreshold) {
       // code here
       i = i + 1;
    }
    return score;
}
```

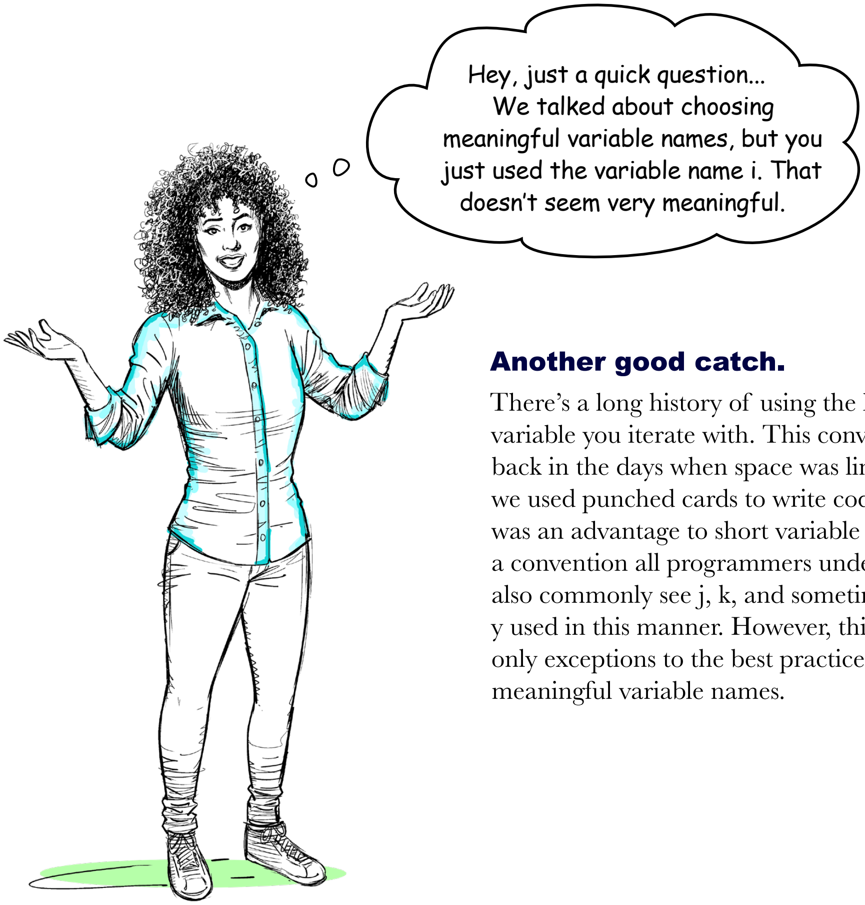
The points, score, and i variables are all declared within a function.

We call them local variables because they are known locally only within the function itself.

Even if we use levelThreshold inside the function, it's global because it's declared outside the function.

But what does it matter? Variables are variables, right? Well, *where* you declare your variables determines *how visible* they are to other parts of your code, and later, understanding how these two kinds of variables operate will help you write more maintainable code (not to mention helping you understand the code of others).

If a variable is declared outside a function, it's GLOBAL. If it's declared inside a function, it's LOCAL.


Hey, just a quick question... We talked about choosing meaningful variable names, but you just used the variable name i. That doesn't seem very meaningful.

Another good catch.

There's a long history of using the letter i as the variable you iterate with. This convention developed back in the days when space was limited (like when we used punched cards to write code), and there was an advantage to short variable names. Now it's a convention all programmers understand. You'll also commonly see j, k, and sometimes even x and y used in this manner. However, this is one of the only exceptions to the best practice of choosing meaningful variable names.

Knowing the scope of your local and global variables

Where you define your variables determines their *scope*; that is, where they're visible to your code and where they aren't. Let's look at an example of both locally and globally scoped variables—remember, the variables you define outside a function are globally scoped, and the function variables are locally scoped.

```
let avatar = "generic";
let skill = 1.0;
let pointsPerLevel = 1000;
let userPoints = 2008;

function getAvatar(points) {
    let level = points / pointsPerLevel;

    if (level == 0) {
       return "Teddy bear";
    } else if (level == 1) {
       return "Cat";
    } else if (level >= 2) {
       return "Gorilla";
    }
}

function updatePoints(bonus, newPoints) {
    let i = 0;
    while (i < bonus) {
        newPoints = newPoints + skill * bonus;
        i = i + 1;
    }
    return newPoints + userPoints;
}

userPoints = updatePoints(2, 100);
avatar = getAvatar(2112);
```

These four variables are globally scoped. That means they are defined and visible in all the code below.

Note that if you link to additional scripts in your page, they will see these global variables, and you will see their global variables too!

The level variable here is local and is visible only to the code within the getAvatar function. That means only this function can access the level variable.

And let's not forget the points parameter, which also has local scope in the getAvatar function.

Note that getAvatar makes use of the pointsPerLevel global variable too.

In updatePoints we have a local variable, i. i is visible to all of the code in updatePoints.

bonus and newPoints are also local to updatePoints, while userPoints is global.

And here in our code we can use only the global variables; we have no access to any variables inside the functions because they're not visible in the global scope.

There's more to the story

Any block can act as the scope of a variable

So far we've looked at global variables—variables declared at the top level—and local variables—variables declared inside functions. But we can declare a variable inside any code block, like the code block of an if statement. So how does scope work in that case? Take a look at the function **`playTurn`**. Here we have three code blocks: a block for the body of the function, a block for the if, and a block for the else. As we'll see, any of these code blocks can define the scope of a variable. Let's take a look:

```
function playTurn(player, location) {
    let points = 0;
    if (location == 1) {
        let message = player +
            ", you just got 100 points!";
        points = points + 100;
        alert(message);
    }

    else {
        points = points + 5;
    }

    return points;
}
let total = playTurn("Jai", 1);
alert("Total: " + total);
```

The player and location parameters are treated as local variables throughout the entire function's code block.

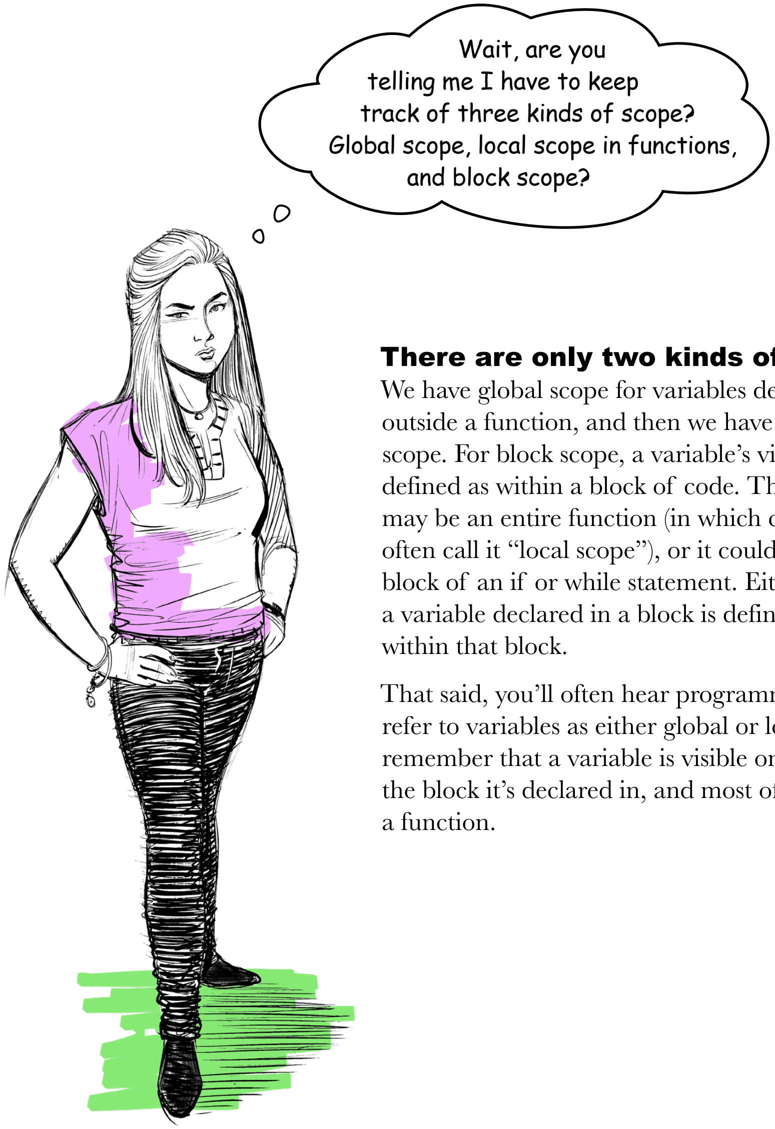
The points variable is also a local variable and visible throughout the entire playTurn function.

The message variable is also a local variable and is visible only in the first block of the if statement.

Note, the message variable is NOT visible in the else block. Also note, the points variable IS, because it's local to the entire function block.

Outside of the function, only global variables, like total, are in scope.

Think of a block of code as any set of statements grouped within curly braces, { and }.


Wait, are you telling me I have to keep track of three kinds of scope? Global scope, local scope in functions, and block scope?

There are only two kinds of scope.
We have global scope for variables declared outside a function, and then we have block scope. For block scope, a variable's visibility is defined as within a block of code. That block may be an entire function (in which case we often call it "local scope"), or it could be the block of an if or while statement. Either way, a variable declared in a block is defined only within that block.

That said, you'll often hear programmers refer to variables as either global or local. Just remember that a variable is visible only within the block it's declared in, and most often, that's a function.

Don't forget to declare your locals!

If you use a variable without declaring it first, that variable will be global. That means that even if you use a variable for the first time inside a function (because you meant for it to be local), the variable will actually be global and be available outside the function too (which might cause confusion later). So, don't forget to declare your locals!

```
function playTurn(player, location) {
   points = 0;
   if (location == 1) {
      message = player + ", you just got 100 points!";
      points = points + 100;
      alert(message);
   }
   else {
      points = points + 5;
   }
   return points;
}
let total = playTurn("Jai", 1);
console.log("Total: " + total);
console.log("Points: " + points);
console.log("Message: " + message);
```

We forgot to declare points with "let" before we used it. So points is **automatically global**.

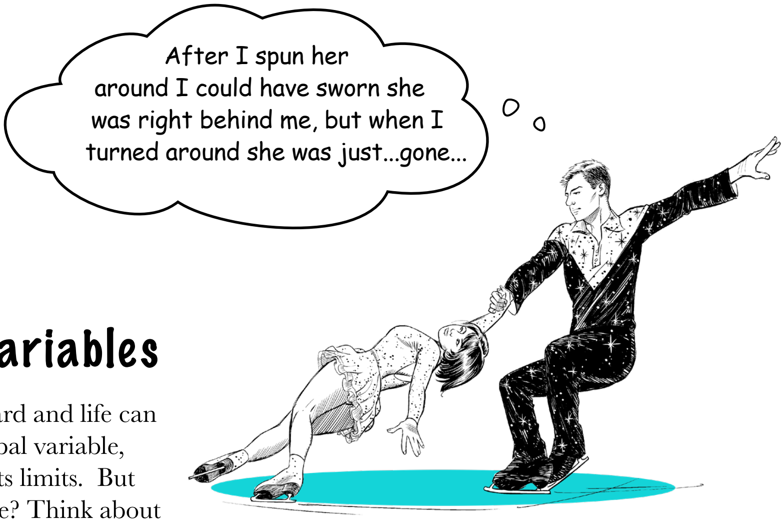
We also forgot to declare message with "let" before we used it. So message is also **automatically global**.

That means we can use points and message outside the function! These variables don't go away (like they should) when the function is done executing.

We shouldn't be able to do this! If you declare these variables properly and you try this code, you'll get an error message in the console: Uncaught ReferenceError: points is not defined.

This is a good thing: it tells you when you're trying to use a variable where it's out of scope. Forgetting to declare your variables can lead to bugs that are difficult to find.

If you forget to declare a variable before using it, the variable will always be global (even if the first time you use it is in a function).


After I spun her around I could have sworn she was right behind me, but when I turned around she was just...gone...

The short lives of variables

When you're a variable, you work hard and life can be short. That is, unless you're a global variable, although even with globals, life has its limits. But what determines the life of a variable? Think about it like this:

Globals live as long as the page. A global variable begins life when its JavaScript is loaded into the page. But your global variable's life ends when the page goes away. Even if you reload the same page, all your global variables are destroyed and then re-created in the newly loaded page.

Local variables typically disappear when your function ends. Local variables are created when your function is first called and live until the function returns (with a value or not). That said, you can take the values of your local variables and return them from the function before the variables meet their digital maker.

We say "typically" because there are some advanced ways to retain locals a little longer, but we're not going to worry about them now.

Local block variables disappear even faster: when the block ends. Local variables in blocks are created when the block begins executing and live until the end of that block (i.e., when you hit the enclosing curly brace, }).

This applies to any statement with one or more blocks, like if/else and while statements and other statements with blocks that you haven't seen yet.

So, there really is NO escape from the page, is there? If you're a local variable, your life comes and goes quickly, and if you're lucky enough to be a global, you're good only as long as the browser doesn't reload the page.

What happens when I name a local variable the same thing as an existing global variable?

You “shadow” your global.

Here’s what that means. Say you have a global variable **`beanCounter`** and you then declare a function, like this:

```
let beanCounter = 10;


function getNumberOfItems(ordertype) {
    let beanCounter = 0;
    if (ordertype == "order") {
       // do some stuff with beanCounter...
    }
    return beanCounter;
}
```

We’ve got a global and a local!

When you do this, any references to **`beanCounter`** within the function refer to the local variable and not the global. So, we say the global variable is *in the shadow* of the local variable (in other words, we can’t see the global variable because the local version is in our way).

Note that the local and global variables have no effect on each other: if you change one, it does not change the other. They are independent variables.

Exercise

Below you'll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify the variables used in all the arguments, parameters, local variables, and global variables. Write the variable names in the appropriate boxes on the right. Then circle any variables that are shadowed. Check your answers at the end of the chapter.

```
let x = 32;
let y = 44;
const radius = 5;

let centerX = 0;
let centerY = 0;
const width = 600;
const height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}
function computeDistance(x1, y1, x2, y2) {
    if (x1 != x2 && y1 != y2) {
        let dx = x1 - x2;
        let dy = y1 - y2;
        let d2 = (dx * dx) + (dy * dy);
        let d = Math.sqrt(d2);
        return d;
    } else {
        return 0;
    }
}
function circleArea(r) {
    let area = Math.PI * r * r;
    return area;
}
setup(width, height);
let area = circleArea(radius);
let distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);
```

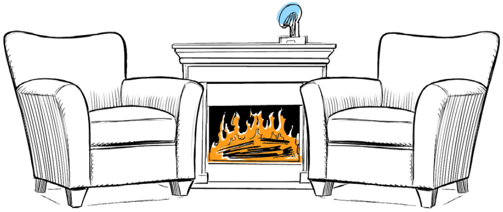
Arguments

Parameters

Local or block

Globals

→ Solution on page 126



Fireside Chats

Tonight's talk: **Global and Local variables argue over who is most important in a program.**

Global variable:	**Local variable:**
Hey Local, I'm really not sure why you're here because I can handle any need for a variable a coder might have. After all, I'm visible everywhere!	Yes, but using globals everywhere is just bad style. Lots of functions need variables that are local. You know, their own private variables for their own use. Globals can be seen everywhere.
You have to admit that I could replace all of you local variables with global ones and the functions would work just the same.	Well, yes and no. If you're extremely careful, sure. But being that careful is difficult, and if you make a mistake, then you've got functions using variables that other functions are using for different purposes. You'd also be littering the program with global variables that you only need inside one function call…that would just make a huge mess.
It wouldn't have to be a mess. Programmers could just create all the variables they need up at the top of a program, so they'd all be in one place…	Yeah, and so what happens if you need to call a function that needs a variable, like, oh I dunno, x, and you can't remember what you've used x for before? You'd have to go searching all over your code to see if you've used x anywhere else! What a nightmare.
Well, if you'd use better names, then you might be able to keep track of your variables more easily.	And what about parameters? Function parameters are always local. So you can't get around that.
True. But why bother with arguments and parameters if you've got all the values you need in globals?	Excuse me, do you hear what you're saying? The whole point of functions is so we can reuse code to compute different things based on different inputs.

Global variable:	**Local variable:**
But local variables are just so...temporary. You come and go at a moment's notice. Actually, it's even worse; you can declare a variable in a tiny block of code like in an if statement. How silly is that?	
	Face it. It's just good programming practice to minimize the visibility of variables to the extent you can...and globals can get you into real trouble. I've seen JavaScript programs that barely use globals at all!
Not at all? Globals are the mainstay of JavaScript programmers!	
	Of inexperienced programmers, sure. But as programmers learn to better structure their code for correctness, maintainability, and just good coding style, they learn to stop using globals except when necessary.
I think I need a drink.	
	They let globals drink? Now, we're *really* in dangerous territory...

there are no Dumb Questions

Q: Keeping track of the scope of all these locals, blocks, and globals is confusing, so why not just stick to globals? That's what I've always done.

A: If you're writing code that is complex or that needs to be maintained over a long period of time, then you really have to watch how you manage your variables. When you're overzealous in creating global variables, it becomes difficult to track where your variables are being used (and where you're making changes to your variables' values), and that can lead to buggy code. All this becomes even more important when you're writing code with coworkers or you're using third-party libraries (although if those libraries are written well, they should be structured to avoid these issues).

So, use globals where it makes sense, but use them in moderation, and whenever possible, make your variables local. As you get more experience with JavaScript, you can investigate additional techniques to structure code so that it's more maintainable.

Q: I have global variables in my page, but I'm loading in other JavaScript files as well. Do those files have separate sets of global variables?

A: There is only one global scope, so every file you load sees the same set of variables (and creates globals in the same space). That's why it is so important you be careful with your use of variables to avoid clashes (and reduce or eliminate global variables when you can). We'll look at other techniques for managing your variables later in the book.

Q: If I use the same name for a parameter as I do for a global variable, does the parameter shadow the global?

A: Yes. Just like if you declare a new local variable in a function with the same name as a global, if you use the same name for a parameter as a global, you're also going to shadow the global with that name. It's perfectly fine to shadow a global name as long as you don't want to use the global variable inside your function. But it's a good idea to document what you're doing with comments so you don't get confused later, when you're reading your code.

Q: Do variables I declare at the top level of my function get shadowed by variables in blocks?

A: Yes. Just like a local variable shadows a global variable, a variable declared inside a block will shadow both a local variable with the same name declared at the top of a function and a global variable with the same name.

Q: If I reload a page in the browser, do the global variables all get reinitialized?

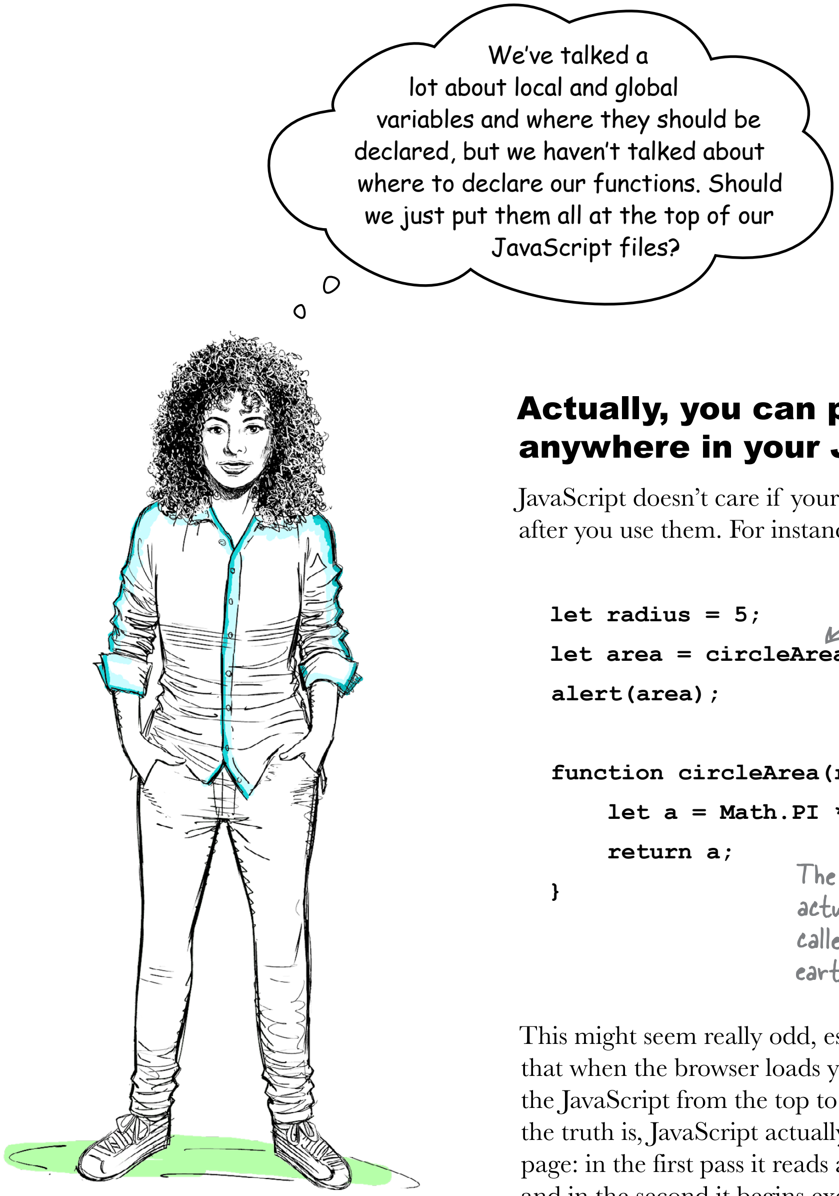
A: Yes. Reloading a page is like starting over from scratch as far as the variables are concerned. And if any code was in the middle of executing when you reload the page, any local variables will disappear, too.

Q: Should we always declare our local variables at the top of a function?

A: Just like with global variables, you can declare local variables when you first need to use them in a function. Or you can go ahead and declare them at the top of your function, so someone reading your code can easily find those declarations and get a sense at a glance of all the local variables used within the function. If you delay declaring a variable and then decide to use that variable earlier in the body of the function than you originally anticipated, you might get behavior that you don't expect. JavaScript creates all local variables at the beginning of a function, whether you declare them or not (this is called "hoisting," and we'll come back to it later), but the variables are all undefined until they are assigned a value, which might not be what you want.

Q: Why does everyone complain about the overuse of global variables in JavaScript?

A: Overusing global variables is an issue when your code reaches a level of complexity where you can easily lose track of how variables are being used across your code. You might find you're reusing a global variable name when you didn't mean to. One of the advantages of JavaScript is that it is easy to jump in and start coding, using global variables. As you learn more, it's a best practice to control the use of global variables, but don't stress over it too much in the beginning. As we said, we'll show you more techniques for dealing with this issue later in the book.


We've talked a lot about local and global variables and where they should be declared, but we haven't talked about where to declare our functions. Should we just put them all at the top of our JavaScript files?

Actually, you can put your functions anywhere in your JavaScript file.

JavaScript doesn't care if your functions are declared before or after you use them. For instance, check out this code:

```
let radius = 5;
let area = circleArea(radius);
alert(area);

function circleArea(r) {
    let a = Math.PI * r * r;
    return a;
}
```

Notice that we're using the circleArea function before we're defining it!

The circleArea function isn't actually defined until after we've called it, in the code above. How on earth does this work?

This might seem really odd, especially if you remember that when the browser loads your page, it starts executing the JavaScript from the top to the bottom of your file. But the truth is, JavaScript actually makes two passes over your page: in the first pass it reads all the function definitions, and in the second it begins executing your code. So, that allows you to place functions anywhere in your file.

The Thing-A-Ma-Jig

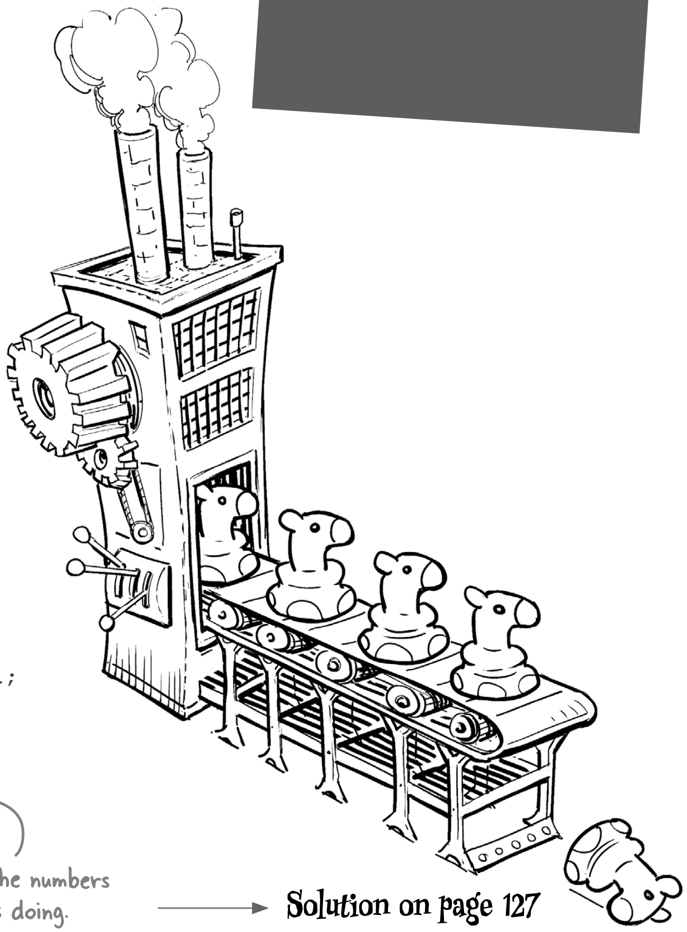
The Thing-A-Ma-Jig is quite a contraption—it clanks and clunks and even thunks, but what it really does, well, you've got us stumped. Coders claim they know how it works. Can you uncrack the code and find its quirks?

```
function clunk(times) {
    let num = times;
    while (num > 0) {
        display("clunk");
        num = num - 1;
    }
}

function thingamajig(size) {
    let facky = 1;
    clunkCounter = 0;
    if (size == 0) {
       display("clank");
    } else if (size == 1) {
       display("thunk");
    } else {
        while (size > 1) {
            facky = facky * size;
            size = size - 1;
        }
        clunk(facky);
    }
}

function display(output) {
    console.log(output);
    clunkCounter = clunkCounter + 1;
}
let clunkCounter = 0;
thingamajig(5);
console.log(clunkCounter);
```

Your output here!

JavaScript console

We recommend passing the Thing-A-Ma-Jig the numbers 0, 1, 2, 3, 4, 5, etc. See if you know what it's doing.



Solution on page 127

Two-Minute Guide to Code Hygiene

There's no place that needs to be well maintained more than your code, and JavaScript can seem pretty loosey-goosey when it comes to organizing your variables and functions. So we've put together a neat little guide for you that makes a few recommendations for those new to coding:

Global variables, right at the TOP!

If you use global variables, and we recommend you don't (once you learn proper techniques not to), it's a good idea to keep your globals grouped together as much as possible, and if they're all up at the top, it's easy to find them. Now, you don't have to do this, but isn't it easier for you and others to locate the variables used in your code if they're all at the top?

Functions like to sit together.

Well, not really; they actually don't care, they're just functions. But if you keep your functions together, it's a lot easier to locate them. As you know, the browser actually scans your JavaScript for the functions before it does anything else, so you can place them at the top or bottom of the file. We often start with our global variables and then put our functions next.

Always declare your variables.

Remember, if you forget to declare your variables inside a function, those variables will be automatically global and may create bugs in your code.

Declare your local variables at the TOP of the function they're in.

Put all your local variable declarations at the beginning of the function body, unless you only need the variable inside one block in the function. This makes them easy to find and ensures they are all declared properly before use.

Use let or const to declare your variables.

For a long time, the only way to declare variables in JavaScript was with `var`. But there were problems with `var`, so a few years ago, JavaScript replaced `var` with `let` and `const`. You don't have to worry about the problems with `var` as long as you always remember to use `let` or `const`.

That's it!

Who am I?

A bunch of JavaScript attendees, in full costume, are playing a party game: "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee.

Tonight's attendees are:

function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, block, Math.random, built-in functions, code reuse

I get passed into a function. ______________________

I send values back to the calling code. ______________________

I'm the all-important keyword. ______________________

I'm what receives arguments. ______________________

I really mean "make a copy." ______________________

I'm everywhere. ______________________

Another phrase for invoking a function. ______________________

Example of a function attached to an object. ______________________

alert and prompt are examples. ______________________

What functions are great for. ______________________

Where I can be seen. ______________________

I'm around when my function is. ______________________

I'm defined within two curly braces. ______________________

→ Solution on page 129

Five-Minute Mystery

The case of the attempted robbery not worth investigating

Sherlock finished his phone call with the bumbling chief of police, Lestrade, and sat down in front of the fireplace to resume reading the newspaper. Watson looked at him expectantly.

"What?" said Sherlock, not looking up from the paper.

"Well? What did Lestrade have to say?" Watson asked.

"Oh, he said they found a bit of rogue code in the bank account where the suspicious activity was taking place."

"And?" Watson said, trying to hide his frustration.

"Lestrade emailed me the code, and I told him it wasn't worth pursuing. The criminal made a fatal flaw and will never be able to actually steal the money," Sherlock said.

"How do you know?" Watson asked.

"It's obvious if you know where to look!" Sherlock exclaimed. "Now stop bothering me with questions and let me finish this paper."

With Sherlock absorbed in the latest news, Watson snuck a peek at Sherlock's phone and pulled up Lestrade's email to look at the code.

```
let balance = 10500;
let cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}

let amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

This is the real, actual bank balance in the account.

Why did Sherlock decide not to investigate the case? How could he know that the criminal would never be able to steal the money just by looking at the code? Is there one problem with the code? Or more?

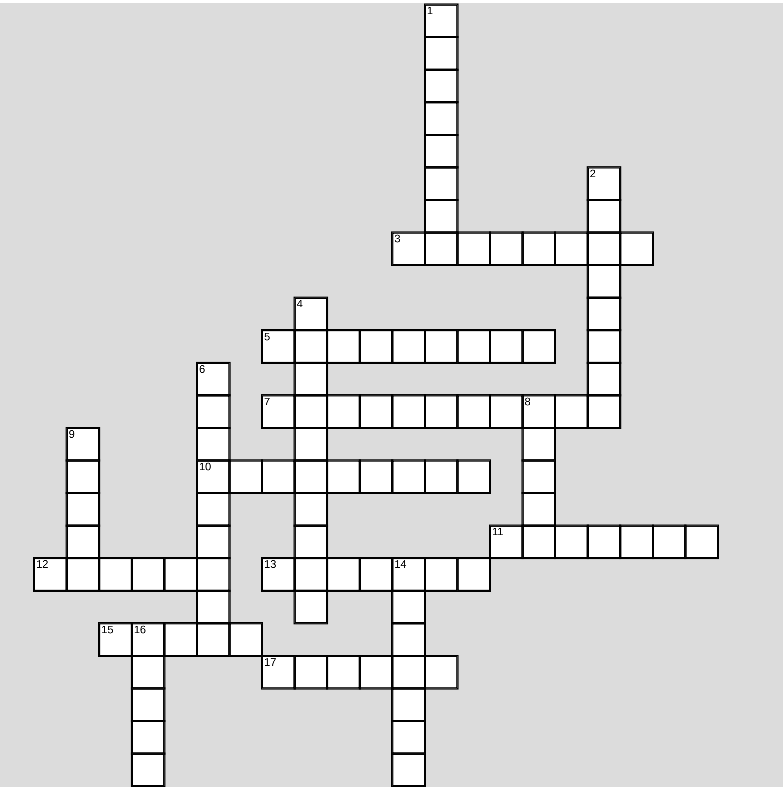
Solution on page 128

Bullet Points

- Declare a function using the **function** keyword, followed by the name of the function.
- Use parentheses to enclose any **parameters** a function has. Use empty parentheses if there are no parameters.
- Enclose the **body** of the function in curly braces.
- The statements in the body of a function are executed when you call the function.
- **Calling** a function and **invoking** a function are the same thing.
- You call a function by using its name and passing arguments to the function's parameters (if any).
- A function can optionally return a value using the **return** statement.
- A function creates a block (or local) scope for parameters and any local variables the function uses.
- Variables are either in the **global scope** (visible everywhere in your program) or in the **local scope** (visible only in the function where they are declared).
- Variables can be local to an entire function or local to just one **block** of code, like an if statement or while statement.
- A **block** of code is a group of statements enclosed in curly braces, like in an if statement or while statement.
- Declare local variables at the top of the body of your function, unless you need the variable in only one block.
- If you forget to declare a local variable using **let** or **const**, that variable will be global, which could have unintended consequences in your program.
- Functions are a good way to organize your code and create reusable chunks of code.
- You can customize the code in a function by passing in arguments to parameters (and using different arguments to get different results).
- Functions are also a good way to reduce or eliminate duplicate code.
- You can use **built-in functions**, like alert, prompt, and Math.random, to do work in your programs.
- Using built-in functions means using existing code you don't have to write yourself.
- It's a good idea to organize your code so your functions are together, and your global variables are together, at the top of your JavaScript file.

JavaScript cross

In this chapter, you got functional. Now use some brain functions to do this crossword.



ACROSS

3. You can declare your functions _______ in your JavaScript file.

5. When you reload your page, all your _______ get reinitialized.

7. JavaScript uses _______ when passing arguments to functions.

10. What gets returned from a function without a return statement.

11. If you forget to declare your locals, they'll be treated like _______.

12. To get a value back from a function, use the _______ statement.

13. _______ through your code means following the execution line by line.

15. Use functions so you can _______ code over and over again.

17. A local variable can _______ a global variable.

DOWN

1. Local variables disappear when the _______ returns.

2. A parameter acts like a _______ in the body of a function.

4. The variables that arguments land in when they get passed to functions.

6. What gets passed to functions.

8. It's better to use _______ variables whenever you can.

9. A variable with global _______ is visible everywhere.

14. Extra arguments to a function are _______.

16. Watson looked at the bank heist code in Sherlock's _______ on his phone.

→ Solution on page 130

Sharpen your pencil Solution

From page 84

Do a little analysis of the code below. How does it look? Choose among the options at the bottom of the page, or write in your own analysis. Here's our solution.

```
let dogName = "rover";
let dogWeight = 23;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
```

We chose all of them!

- ☑ A. The code seems very redundant.
- ☑ B. If we wanted to change how this outputs, or if we wanted to add another weight for dogs, this is going to require a lot of reworking.
- ☑ C. Looks tedious to type in!
- ☑ D. Not the most readable code I've ever seen.
- ☑ E. Looks like the developer thought the weights might change over time.

Sharpen your pencil Solution

From page 90

We've got some more calls to bark below. Next to each call, write what you think the output should be, or if you think the code will cause an error. Here's our solution.

`bark("juno", 20);` juno says woof woof

`bark("scottie", -1);` scottie says woof woof

Our bark function doesn't check to make sure dog weights are greater than 0. So this works because -1 is less than 20.

`bark("dino", 0, 0);` dino says woof woof

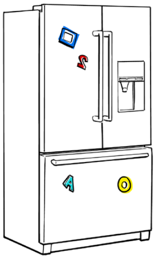
The bark function just ignores the extra argument, 0. And using 0 as the weight doesn't make sense, but it still works.

`bark("fido", "20");` fido says woof woof

We compare the string "20" to the number 20. "20" isn't greater than 20, so fido says "woof woof." (You'll find out later how JavaScript compares "20" and 20.)

`bark("lady", 10);` lady says woof woof

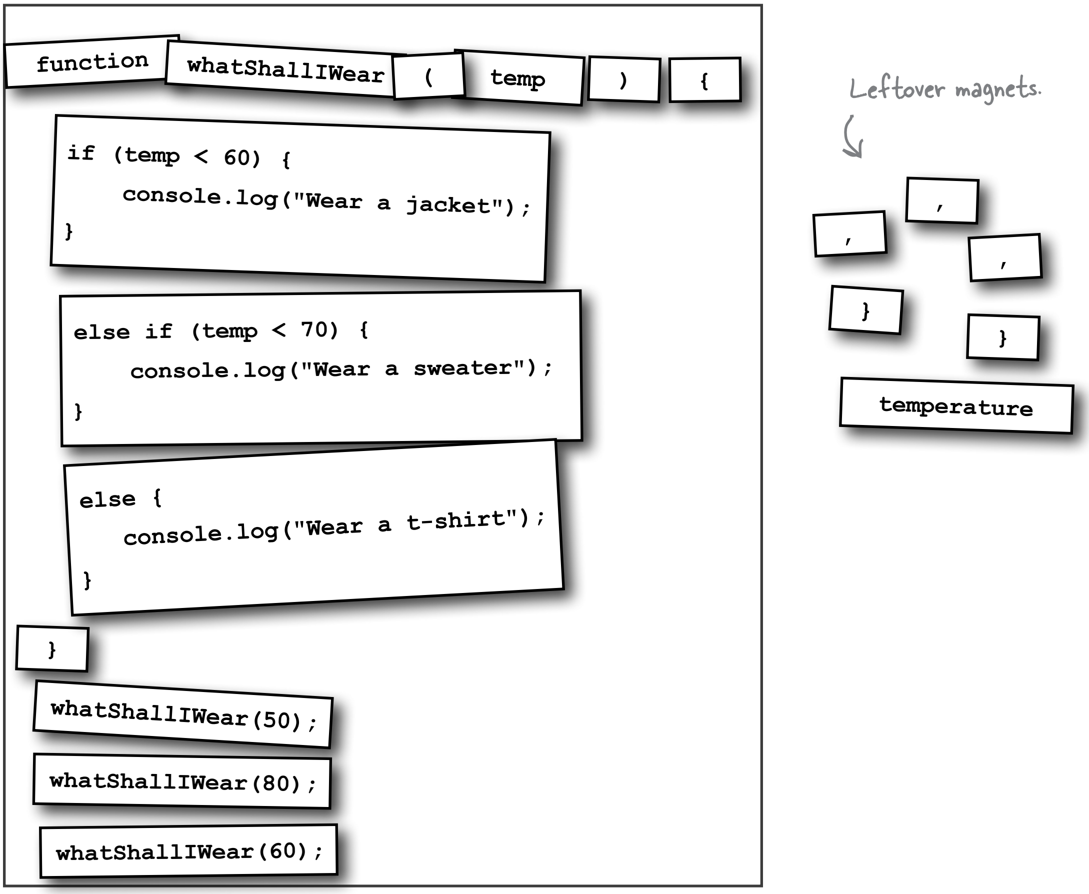
`bark("bruno", 21);` bruno says WOOF WOOF



Code Magnets Solution

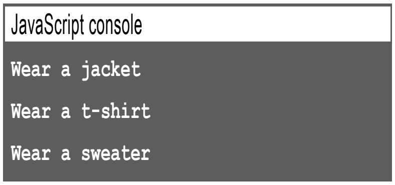
From page 91

This working JavaScript code is all scrambled up on the fridge. Can you reorder the code snippets to make a working program that produces the output listed below? Note that there may be some extra code on the fridge, so you may not use all the magnets. Here's our solution.



```
function whatShallIWear ( temp ) {
  if (temp < 60) {
      console.log("Wear a jacket");
  }
  else if (temp < 70) {
      console.log("Wear a sweater");
  }
  else {
      console.log("Wear a t-shirt");
  }
}
whatShallIWear(50);
whatShallIWear(80);
whatShallIWear(60);
```

Leftover magnets.



JavaScript console

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

Exercise Solution

From page 95

Below you'll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify all the variables and constants, functions, arguments, and parameters. Write the names of each in the appropriate boxes on the right. Here's our solution.

```
function dogYears(dogName, age) {
    let years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
const myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
let guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    const defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
let pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables and constants

myDog, guests, pet, years, defaultSound

Functions

dogYears, makeTea, secret, speak

Built-in functions

alert, console.log, prompt

Arguments

myDog, 4, guests, "Earl Grey", pet, plus all the string arguments to alert and console.log

Parameters

dogName, age, cups, tea, kind

Exercise Solution

From page 111

Below you'll find some JavaScript code, including variables, function definitions, and function calls. Your job is to identify the variables used in all the arguments, parameters, local or block variables, and global variables. Write the variable names in the appropriate boxes on the right. Then circle any variables that are shadowed. Here's our solution.

```
let x = 32;
let y = 44;
const radius = 5;

let centerX = 0;
let centerY = 0;
const width = 600;
const height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}
function computeDistance(x1, y1, x2, y2) {
    if (x1 != x2 && y1 != y2) {
        let dx = x1 - x2;
        let dy = y1 - y2;
        let d2 = (dx * dx) + (dy * dy);
        let d = Math.sqrt(d2);
        return d;
    } else {
        return 0;
    }
}
function circleArea(r) {
    let area = Math.PI * r * r;
    return area;
}
setup(width, height);
let area = circleArea(radius);
let distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);
```

Arguments

Don't forget the arguments to the alert function.

width, height, radius, x, y, centerX, centerY, "Area: " + area, "Distance: "+ distance

Parameters

The parameters width and height shadow the global variables width and height.

width, height, x1, y1, x2, y2, r

Locals or block

The local variable area shadows the global variable area.

dx, dy, d2, d, area

Globals

Don't forget area and distance. These are globals too.

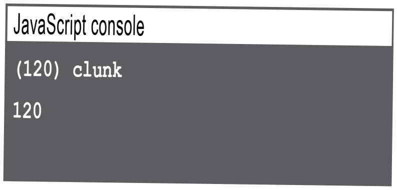
x, y, radius, centerX, centerY, width, height, area, distance

From page 116

The Thing-A-Ma-Jig

The Thing-A-Ma-Jig is quite a contraption—it clanks and clunks and even thunks, but what it really does, well, you've got us stumped. Coders claim they know how it works. Can you uncrack the code and find its quirks?

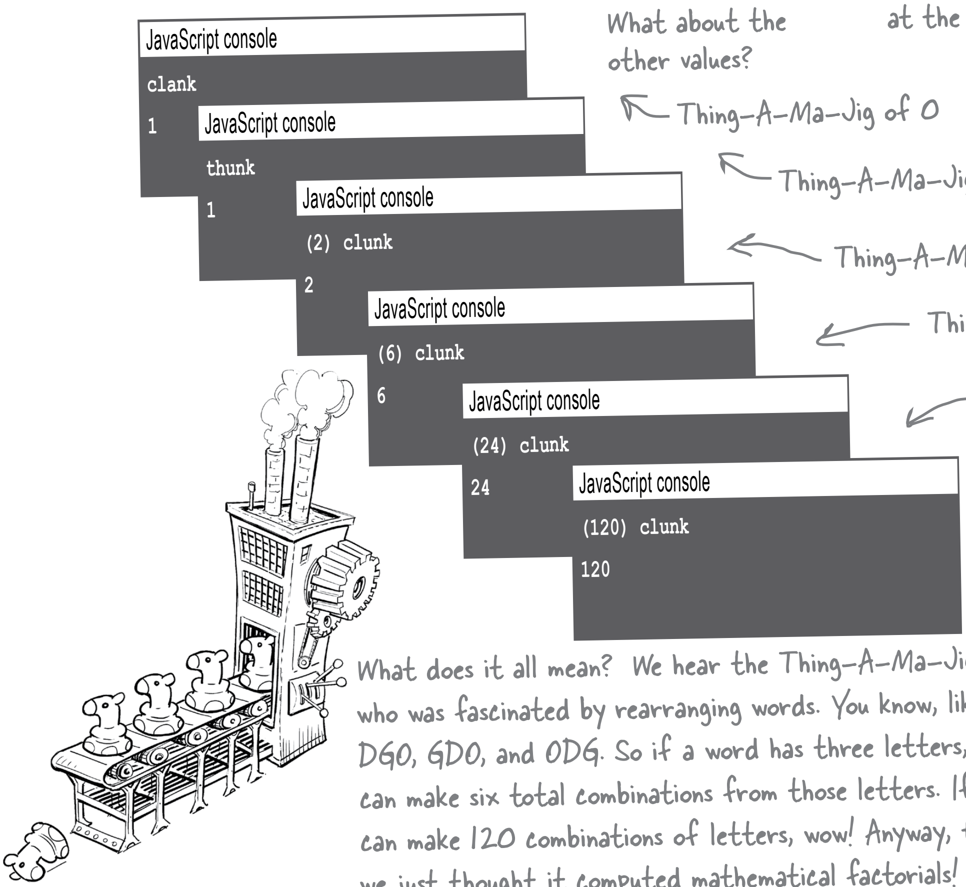
Here's our solution.



```
JavaScript console
(120) clunk
120
```

If you pass 5 to the thingamajig function, you'll see "clunk" in the console 120 times (or you might see your console abbreviate 120 clunks, like this), and then the number 120 at the end.

What about the other values?



Thing-A-Ma-Jig of 0

```
JavaScript console
clank
1
```

Thing-A-Ma-Jig of 1

```
JavaScript console
thunk
1
```

Thing-A-Ma-Jig of 2

```
JavaScript console
(2) clunk
2
```

Thing-A-Ma-Jig of 3

```
JavaScript console
(6) clunk
6
```

Thing-A-Ma-Jig of 4

```
JavaScript console
(24) clunk
24
```

Thing-A-Ma-Jig of 5

```
JavaScript console
(120) clunk
120
```

What does it all mean? We hear the Thing-A-Ma-Jig was invented by a curious chap who was fascinated by rearranging words. You know, like DOG rearranged is GOD, OGD, DGO, GDO, and ODG. So if a word has three letters, the Thing-A-Ma-Jig says you can make six total combinations from those letters. If you use the word "mixes" you can make 120 combinations of letters, wow! Anyway, that's what we heard. And here we just thought it computed mathematical factorials! Who knew!?

Check out "factorial" on Wikipedia for more info!

From page 119

Five Minute Mystery Solution

Why did Sherlock decide not to investigate the case? How could he know that the criminal would never be able to steal the money just by looking at the code? Is there one problem with the code? Or more? Here's our solution.

```
let balance = 10500;
let cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}
```

balance is a global variable...

...but it's shadowed by this parameter.

So when you change the balance in the function steal, you're not changing the actual bank balance!

We're returning the amount stolen...

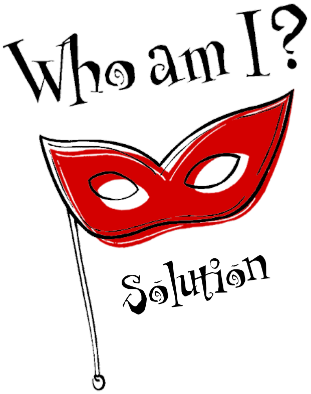
```
let amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

...but we're not using it to update the real balance in the account. So the balance of the bank account is the same as it was originally.

The criminal thinks he stole the money, but he didn't!

And, in addition to not actually stealing any money, the criminal forgets to turn the camera back on, which is a dead giveaway to the police that something nefarious is going on. Remember, when you return from a function, the function stops executing, so any lines of code after the return are ignored!

From page 118



A bunch of JavaScript attendees, in full costume, are playing a party game: "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee. Here's our solution.

Tonight's attendees are:

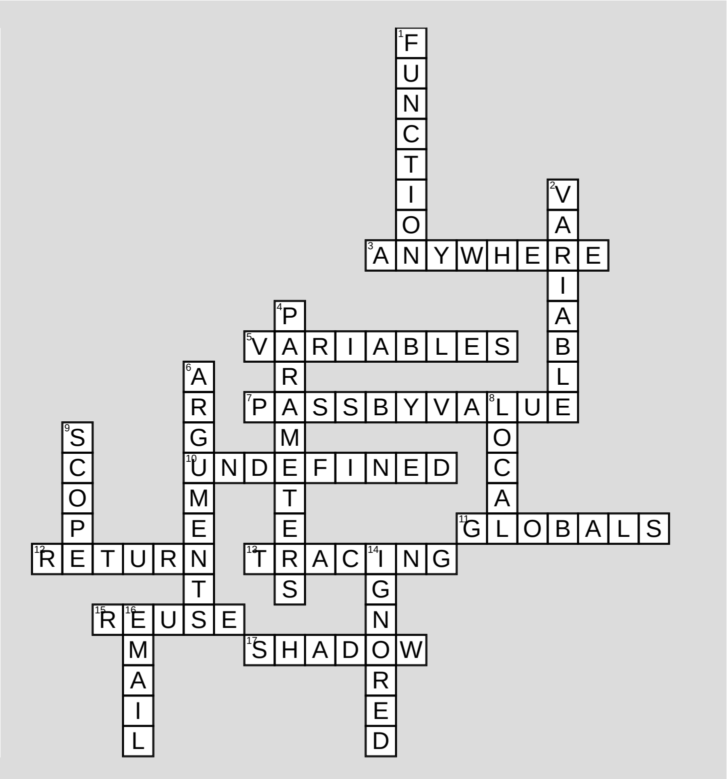
function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, block, Math.random, built-in functions, code reuse

I get passed into a function.	argument
I send values back to the calling code.	return
I'm the all-important keyword.	function
I'm what receives arguments.	parameter
I really mean "make a copy."	pass-by-value
I'm everywhere.	global variable
Another phrase for invoking a function.	function call
Example of a function attached to an object.	Math.random
alert and prompt are examples.	built-in functions
What functions are great for.	code reuse
Where I can be seen.	scope
I'm around when my function is.	local variable
I'm defined within two curly braces .	block

JavaScript cross Solution

From page 121

In this chapter, you got functional. Now use some brain functions to do this crossword. Here's our solution.

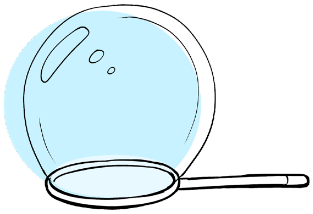

1 FUNCTION
2 VARIABLE
3 ANYWHERE
4 PARAMETERS
5 VARIABLES
6 ARGUMENTS
7 PASSBYVALUE
8 LOCAL
9 SCOPE
10 UNDEFINED
11 GLOBALS
12 RETURN
13 TRACING
14 IGNORED
15 REUSE
16 EMAIL
17 SHADOW

4 putting some order in your data

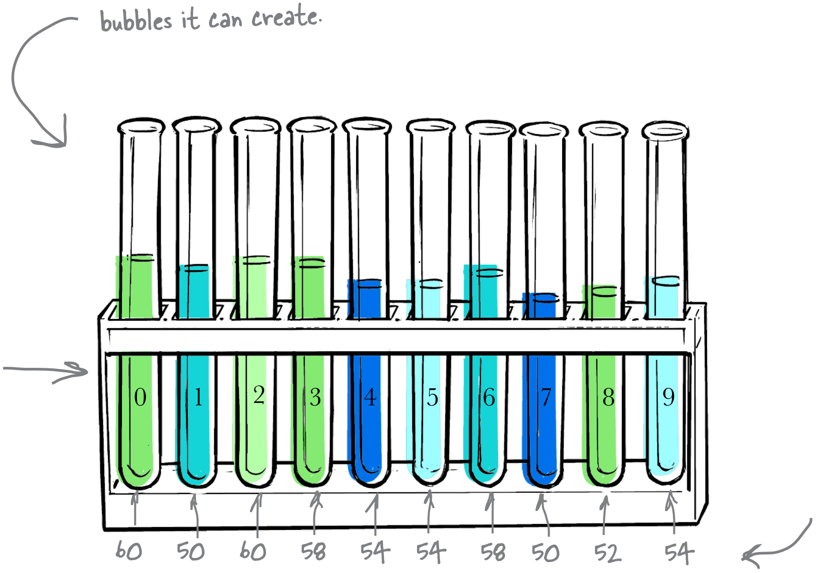
Arrays

There's more to JavaScript than numbers, strings, and booleans. So far you've been writing JavaScript code with **primitives**—strings, numbers, and booleans, like "Fido", 23, and true. You can do a lot with primitive types, but at some point you've got to deal with **more data**. Say, all the items in a shopping cart, or all the songs in a playlist, or a set of stars and their apparent magnitude, or an entire product catalog. For that you need a little more *oomph*. The type of choice for this kind of ordered data is a JavaScript **array**, and in this chapter we're going to walk through how to put your data into an array, how to pass it around, and how to operate on it. We'll be looking at a few other ways to **structure your data** in later chapters, but let's get started with arrays.

Can you help Bubbles-R-Us?



Meet the Bubbles-R-Us company. Their tireless research makes sure bubble wands and machines everywhere blow the best bubbles. Today, they're testing the "bubble factor" of several variants of their new bubble solution; that is, they're testing how many bubbles a given solution can make. Here's their data:



Each bubble solution was tested for the number of bubbles it can create.

Each test tube is labeled 0 to 9 and contains a slightly different bubble solution.

And here's the bubble factor score for each solution.

Of course, you want to get all this data into JavaScript so you can write code to help analyze it. But that's a lot of values. How are you going to construct your code to handle all these values?

How to represent multiple values in JavaScript

You know how to represent single values like strings, numbers, and booleans with JavaScript, but how do we represent *multiple* values, like all the bubble factor scores from the ten bubble solutions? To do that, we use JavaScript *arrays*. An array is a JavaScript type that can hold many values. Here's a JavaScript array that holds all the bubble factor scores:

```
let scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54];
```

Here are all ten values, grouped together into an array and assigned to the scores variable.

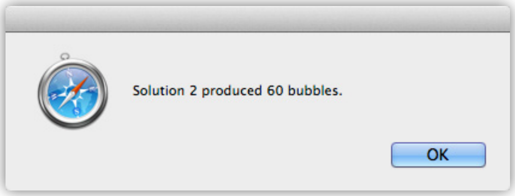
When we define an array like this, we say we're creating an array literal, because we're literally writing out the values.

You can treat all the values as a whole, or you can access the individual scores when you need to. Check this out:

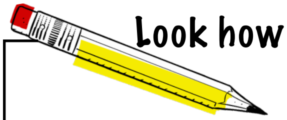
To access an item of the array we use this syntax: the variable name of the array followed by the index of the item, surrounded by square brackets.

Notice that arrays are zero-based. So, the first bubble solution is solution #0 and has the score in scores[0], and likewise, the third bubble solution is solution #2 and has the score in scores[2].

```
let solution2 = scores[2];
alert("Solution 2 produced " + solution2 + " bubbles.");
```



Solution 2 produced 60 bubbles.



Look how easy it is to create arrays

You don't know much about arrays yet, but we bet you can make some good guesses about how they work. Take a look at each line of code below and see if you can guess what it does. Write in your answers, and check our solution before you continue.

```
let testResults = [91, 84, 100];
let dogs = ["Fido", "Spot"];
let isBarking = [true, false];

let dogBarking1 = isBarking[0];
if (dogBarking1 == true) {
    console.log(dogs[0] + " is barking");
}

if (testResults[2] > 90) {
    console.log("You got an A!");
}
```

Create an array of integer test results.

Solution on page 175

How arrays work

Before we get on to helping Bubbles-R-Us, let's make sure we've got arrays down. As we said, you can use arrays to store *multiple* values (unlike variables that hold just one value, like a number or a string). Most often you'll use arrays when you want to group together similar things, like bubble factor scores, ice cream flavors, daytime temperatures, or even the answers to a set of true/false questions. Once you have a bunch of values you want to group together, you can create an array that holds them, and then access those values in the array whenever you need them.

How to create an array

Let's say you wanted to create an array that holds ice cream flavors. Here's how you'd do that:

Notice that each item in the array is separated by a comma.

```
let flavors = ["vanilla", "butterscotch", "lavender", "chocolate", "cookie dough"];
```

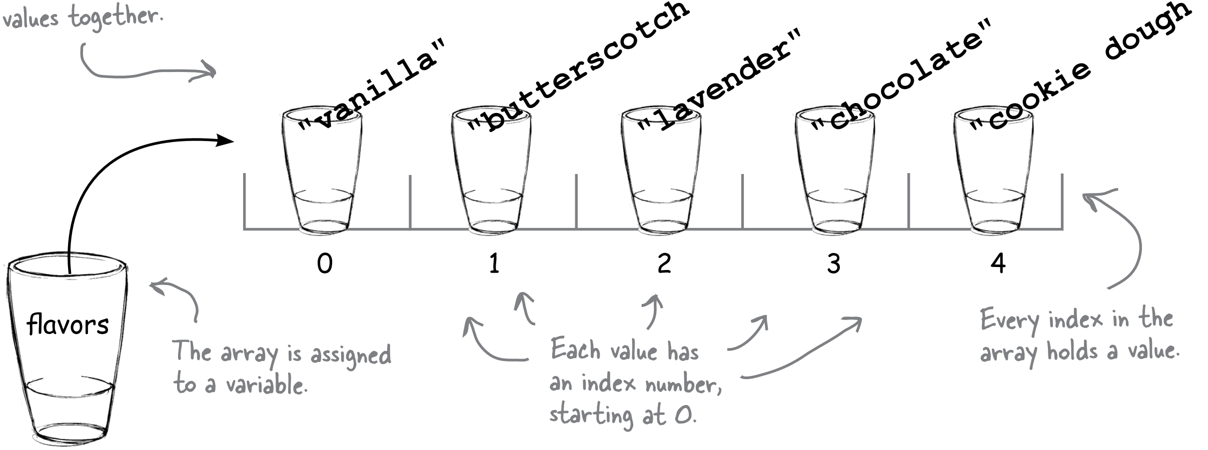
Let's assign the array to a variable named flavors.

To begin the array, use the [character...

...then list each item of the array...

...and end the array with the] character.

When you create an array, each item is placed at a location, or *index*, in the array. With the flavors array, the first item, "vanilla", is at index 0, the second, "butterscotch", is at index 1, and so on. Here's a way to think about an array:



The array collects all these values together.

The array is assigned to a variable.

Each value has an index number, starting at 0.

Every index in the array holds a value.

How to access an array item

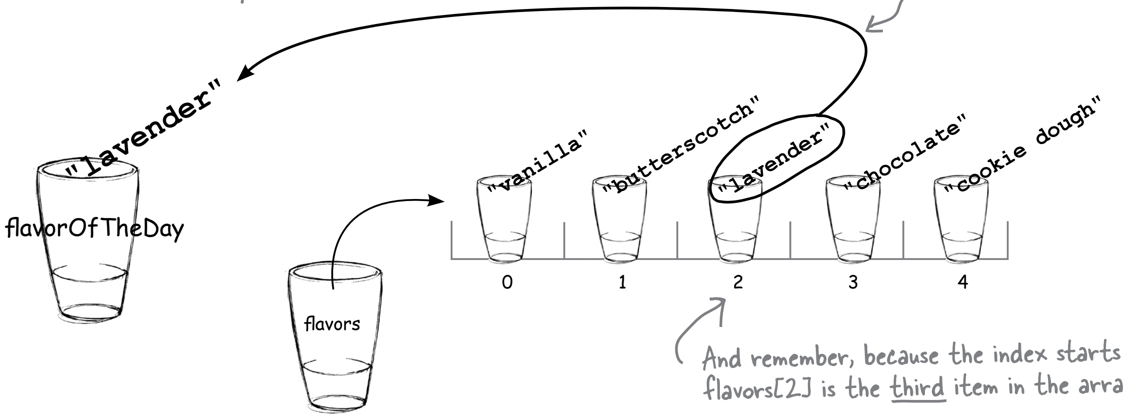
Each item in the array has its own index, and that's your key to both accessing and changing the values in an array. To access an item, just follow the array variable name with an index, surrounded by square brackets. You can use that notation anywhere you'd use a variable:

```
let flavorOfTheDay = flavors[2];
```

This evaluates to the value of the flavors array at index 2 (which is "lavender"), which is then assigned to the variable flavorOfTheDay.

To get an element from an array, you need both the name of the array and the index of the value you want.

flavorOfTheDay is assigned the value in flavors[2].


"lavender"
flavorOfTheDay
flavors
"vanilla"
"butterscotch"
"lavender"
"chocolate"
"cookie dough"
0
1
2
3
4

And remember, because the index starts at 0, flavors[2] is the third item in the array.

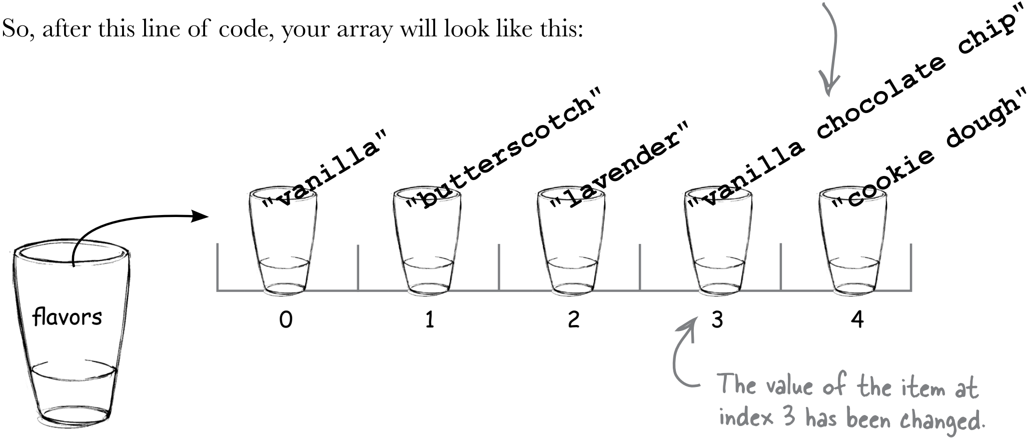
Updating a value in the array

You can also use the array index to change a value in an array:

```
flavors[3] = "vanilla chocolate chip";
```

This sets the value of the item at index 3 (previously "chocolate") to a new value, "vanilla chocolate chip".

So, after this line of code, your array will look like this:


flavors
"vanilla"
"butterscotch"
"lavender"
"vanilla chocolate chip"
"cookie dough"
0
1
2
3
4

The value of the item at index 3 has been changed.

How big is that array anyway?

Say someone hands you a nice big array with important data in it. You know what's in it, but you probably won't know exactly how big it is. Luckily, every array comes with its own property, `length`. We'll talk more about properties and how they work in the next chapter, but for now, a property is just a value associated with an array. Here's how you use the `length` property:

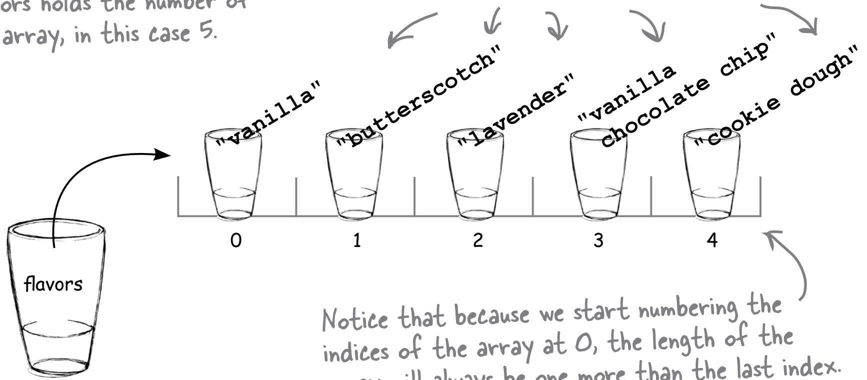
Every array has a property, length, that contains the number of items currently in the array.

To get the length of an array, you use the name of the array, then a ".", and then length.

```
let numFlavors = flavors.length;
```

Now numFlavors holds the number of items in the array, in this case 5.

The length is 5 because there are 5 items.



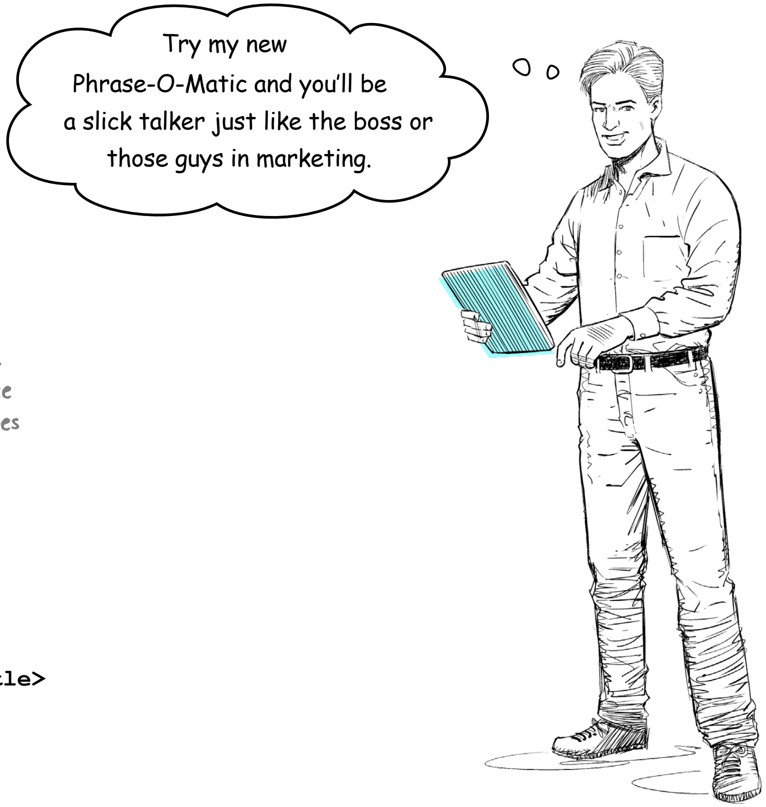
Notice that because we start numbering the indices of the array at 0, the length of the array will always be one more than the last index.

Sharpen your pencil

The products array below holds the Jenn and Berry's ice cream flavors. The ice cream flavors were added to this array in the order of their creation. Finish the code to determine the *most recent* ice cream flavor they created.

```
let products = ["Choo Choo Chocolate", "Icy Mint", "Cake Batter", "Bubblegum"];
let last = ________________;
let recent = products[last];
```

→ Solution on page 175


Try my new Phrase-O-Matic and you'll be a slick talker just like the boss or those guys in marketing.

Check out this code for the hot new Phrase-O-Matic app and see if you can figure out what it does before you go on...

```
<!doctype html>
<html lang="en">
<head>
  <title>Phrase-O-Matic</title>
  <meta charset="utf-8">
  <script>
    function makePhrases() {
      let words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
      let words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
      let words3 = ["process", "solution", "tipping point", "strategy", "vision"];

      let rand1 = Math.floor(Math.random() * words1.length);
      let rand2 = Math.floor(Math.random() * words2.length);
      let rand3 = Math.floor(Math.random() * words3.length);

      let phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
      alert(phrase);
    }
    makePhrases();
  </script>
</head>
<body></body>
</html>
```

You didn't think our serious business application from Chapter 1 was serious enough? Fine. Try this one, if you need something to show the boss.

The Phrase-O-Matic

We hope you figured out this code is the perfect tool for creating your next start-up marketing slogan. It has created winners like "B-to-B value-added strategy" and "24/7 empowered process" in the past, and we have high hopes for more winners in the future. Let's see how this thing really works:

① First, we define the `makePhrases` function, which we can call as many times as we want to generate the phrases we want.

We're defining a function named makePhrases that we can call later.

```
function makePhrases() {

}
makePhrases();
```

All the code for makePhrases goes here; we'll get to it in a sec...

We call makePhrases once here, but we could call it multiple times if we wanted more than one phrase.

② With that out of the way, we can write the code for the `makePhrases` function. Let's start by setting up three arrays. Each will hold words that we'll use to create the phrases. In the next step, we'll pick one word at random from each array to make a three-word phrase.

We create a variable named words1 that we can use for the first array.

```
let words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
```

We're putting five strings in the array. Feel free to change these to the latest buzzwords out there.

```
let words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
let words3 = ["process", "solution", "tipping point", "strategy", "vision"];
```

And here are two more arrays of words, assigned to two new variables, words2 and words3.

③ Now we generate three random numbers, one for each of the three random words we want to pick to make a phrase. Remember from Chapter 2 that Math.random generates a number between 0 and 1 (not including 1). If we multiply that by the length of the array and use Math.floor to truncate the number, we get a number between 0 and one less than the length of the array.

```
let rand1 = Math.floor(Math.random() * words1.length);
let rand2 = Math.floor(Math.random() * words2.length);
let rand3 = Math.floor(Math.random() * words3.length);
```

rand1 will be a number between 0 and the last index of the words1 array.

And likewise for rand2 and rand3.

④ Now we create the slick marketing phrase by taking each randomly chosen word and concatenating them all together, with nice spaces in between for readability.

We define another variable to hold the phrase.

We use each random number to index into the word arrays.

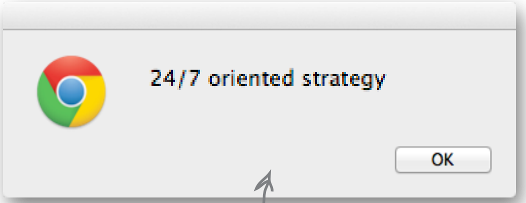
```
let phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
```

⑤ We're almost done; we have the phrase, now we just have to display it. We're going to use `alert` (but you could also use `console.log` and check your browser's console window).

```
alert(phrase);
```

⑥ Okay, finish that last line of code, have one more look over it all, and feel that sense of accomplishment before you load it into your browser. Give it a test drive and enjoy the phrases.

Here's what ours looks like!



24/7 oriented strategy

Just reload the page for endless start-up possibilities (okay, not endless, but work with us here, we're trying to make this simple code exciting!).

there are no Dumb Questions

Q: Does the order of items in an array matter?

A: Most of the time, yes, but it depends. In the Bubbles-R-Us scores array, the ordering matters a lot, because the index of the score in the array tells us which bubble solution got that score—bubble solution 0 got score 60, and that score is stored at index 0. If we mixed up the scores in the array, then we'd ruin the experiment! However, in other cases, the order may not matter. For instance, if you're using an array just to keep a list of randomly selected words and you don't care about the order, then it doesn't matter which order they're in in the array. But if you later decide you want the words to be in alphabetical order, then the order will matter. So it really depends on how you're using the array. You'll probably find that ordering matters more often than not when you use an array.

Q: How many things can you put into an array?

A: Theoretically, as many as you want. Practically, however, the number is limited by the memory on your computer. Each array item takes up a little bit of space in memory. Remember that JavaScript runs in a browser, and that browser is one of many programs running on your computer. If you keep adding items to an array, eventually you'll run out of memory space. However, depending on the kinds of items you're putting in your array, the maximum number of items you can put into an array is probably in the many thousands, if not millions, which you're unlikely to need most of the time. And keep in mind that the more items you have the slower your program will run, so you'll want to limit your arrays to reasonable sizes—say, a few hundred—most of the time.

Q: Can you have an empty array?

A: You can, and in fact, you'll see an example of using an empty array shortly. To create an empty array, just write:

```
let emptyArray = [ ];
```

If you start with an empty array, you can add things to it later.

Q: So far we've seen strings and numbers in an array; can you put other things in arrays too?

A: You can; in fact, you can put just about any value you'll find in JavaScript in an array, including numbers, strings, booleans, other arrays, and even objects (we'll get to this later).

Q: Do all the values in an array have to be the same type?

A: No, they don't, although typically we *do* make the values all of the same type. Unlike in many other languages, there is no requirement in JavaScript that all the values in an array be of the same type. However, if you mix up the types of the values in an array, you need to be extra careful when using those values. Here's why: let's say you have an array with the values [1, 2, "fido", 4, 5]. If you then write code that checks to see if the values in the array are greater than, say, 2, what happens when you check to see if "fido" is greater than 2? To make sure you aren't doing something that doesn't make sense, you'd have to check the type of each of the values before you used it in the rest of your code. It's certainly possible to do this (and you'll see how later in the book), but in general, it's a lot easier and safer if you just use the same type for all the values in your arrays.

Q: What happens if you try to access an array with an index that is too big or too small (like less than 0)?

A: If you have an array, like this one:

```
let a = [1, 2, 3];
```

and you try to access a[10] or a[-1], in either case, you'll get the result undefined. So, either you'll want to make sure you're using only valid indices to access items in your array, or you'll need to check that the value you get back is not undefined.

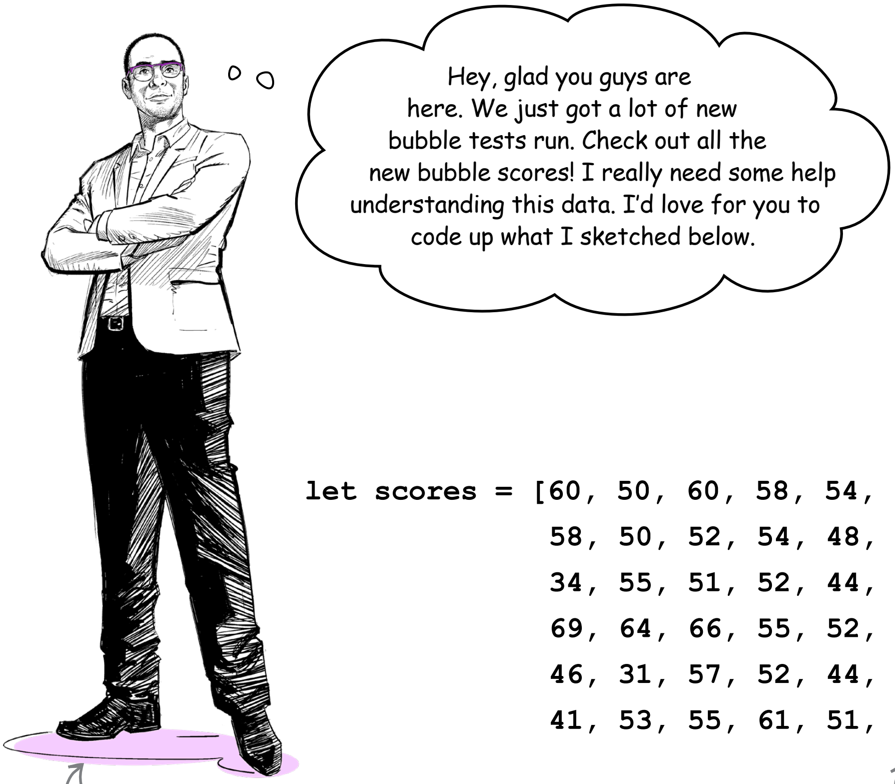
Q: So, I can see how to get the first item in an array using index 0. But how would I get the last item in an array? Do I always have to know precisely how many items are in my array?

A: You can use the length property to get the last item of an array. You know that length is always one greater than the last index of the array, right? So, to get the last item in the array, you can write:

```
myArray[myArray.length - 1];
```

JavaScript gets the length of the array, subtracts one from it, and then gets the item at that index number. So if your array has 10 items, it will get the item at index 9, which is exactly what you want. You'll use this trick all the time to get the last item in an array when you don't know exactly how many items are in it.

Meanwhile, back at Bubbles-R-Us...

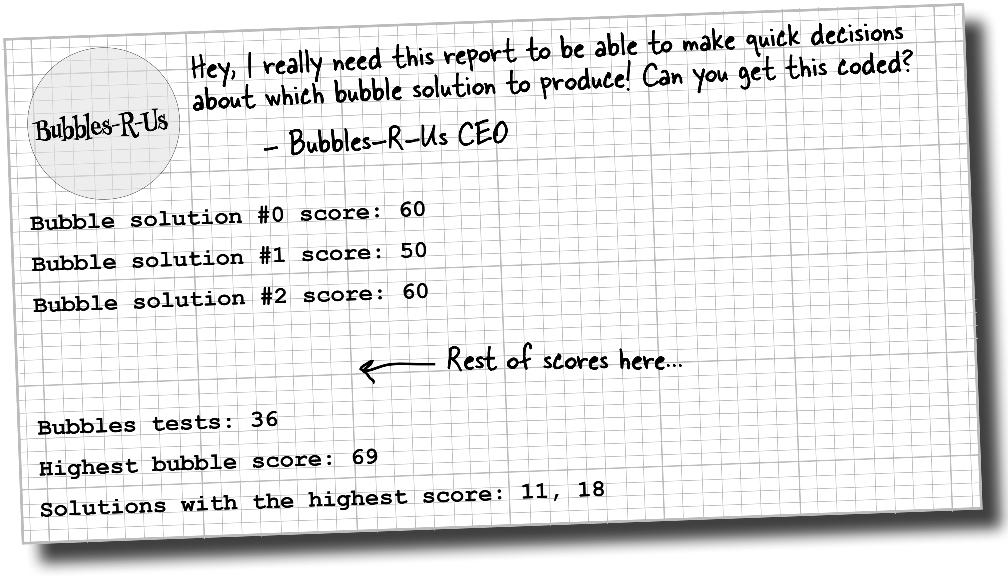

Hey, glad you guys are here. We just got a lot of new bubble tests run. Check out all the new bubble scores! I really need some help understanding this data. I'd love for you to code up what I sketched below.

```
let scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];
```

The Bubbles-R-Us CEO

What we need to build

New bubble scores



Bubbles-R-Us

Hey, I really need this report to be able to make quick decisions about which bubble solution to produce! Can you get this coded?

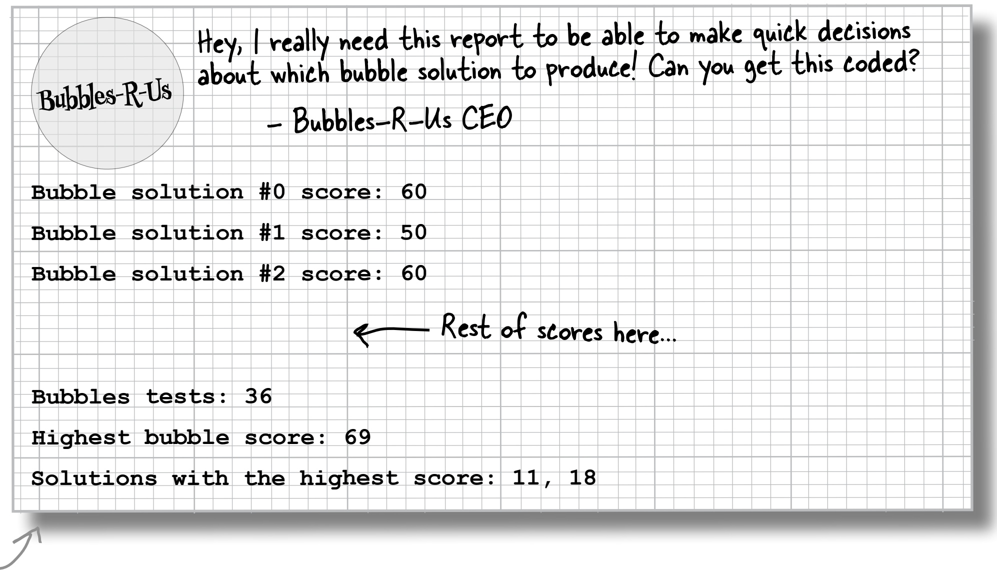
– Bubbles-R-Us CEO

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
```

Rest of scores here...

```
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11, 18
```

Let's take a closer look at what the CEO is looking for:



Bubbles-R-Us

Hey, I really need this report to be able to make quick decisions about which bubble solution to produce! Can you get this coded?

– Bubbles-R-Us CEO

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60

Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11, 18
```

Rest of scores here...

We need to start by listing all the solutions and their corresponding scores.

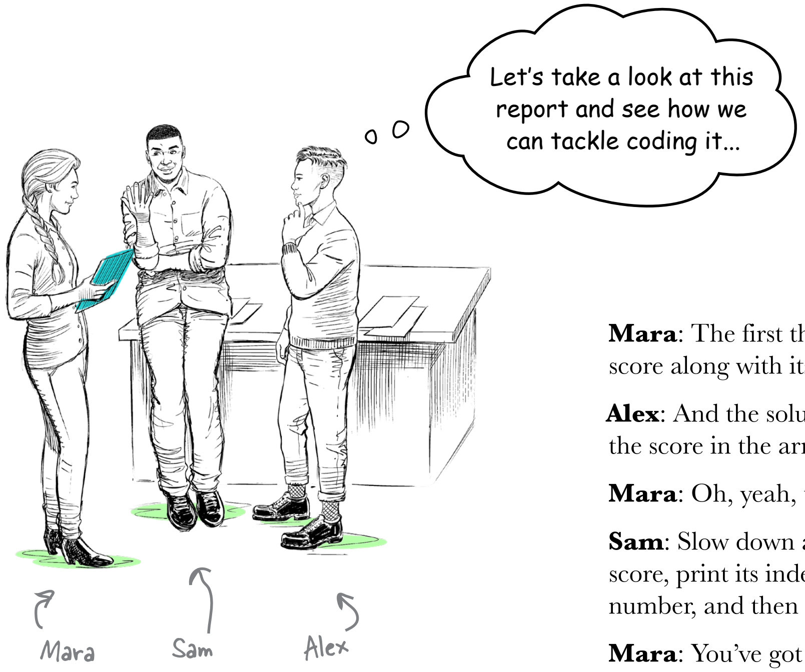
Then we need to print the total number of bubble scores...

...followed by the highest score and each solution that has that score.

Brain Power

Take some time to sketch out your ideas of how you'd create this little bubble score report. Take each item in the report separately and think about how you'd break it down and generate the right output. Make your notes here.

Cubicle Conversation


Let's take a look at this report and see how we can tackle coding it...
Mara
Sam
Alex

Mara: The first thing we need to do is display every score along with its solution number.

Alex: And the solution number is just the index of the score in the array, right?

Mara: Oh, yeah, that's totally right.

Sam: Slow down a sec. So we need to take each score, print its index, which is the bubble solution number, and then print the corresponding score.

Mara: You've got it, and the score is just the corresponding value in the array.

Alex: So, for bubble solution #10, its score is just `scores[10]`.

Mara: Right.

Sam: Okay, but there are a lot of scores. How do we write code to output all of them?

Mara: Iteration, my friend.

Sam: Oh, you mean like a while loop?

Mara: Right, we loop through all the values from zero to the length...oh, I mean the length minus one, of course.

Sam: This is starting to sound very doable. Let's write some code; I think we know what we're doing.

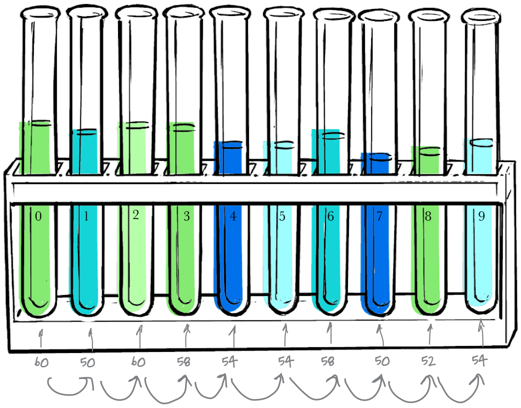
Mara: That works for me! Let's do it, and then we'll come back to the rest of the report.

How to iterate over an array

Your goal is to produce some output that looks like this:

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
.
.
.
Bubble solution #35 score: 44
```

Scores 3 through 34 will be here...we're saving some trees (or bits, depending on which version of the book you have).



We'll do that by outputting the score at index zero, and then we'll do the same for indices one, two, three and so on, until we reach the last index in the array. You already know how to use a while loop. Let's see how we can use that to output all the scores:

And then we'll show you a better way in a sec...

```
let scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

let i = 0;

while (i < scores.length) {

    let output = "Bubble solution #" + i + " score: " + scores[i];

    console.log(output);

    i = i + 1;

}
```

Create a variable to keep track of the current index.

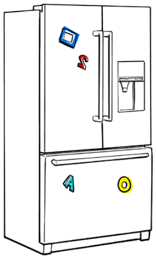
And keep looping while the index is less than the length of the array.

We're using this variable to create a string to output.

Then, create a string to use as a line of output that includes the bubble solution number (which is just the array index) and the score.

Next, use console.log to output the string.

And finally, increment the index by one before looping again.



Code Magnets

We've got code for testing to see which ice cream flavors have bubblegum pieces in them. We had all the code nicely laid out on our fridge using fridge magnets, but the magnets fell on the floor. It's your job to put them back together. Be careful; a few extra magnets got mixed in. Check your answer at the end of the chapter before you go on.

```
while (i < hasBubbleGum.length)
```

```
i = i + 2;
```

```
{
```

```
}
```

```
i = i + 1;
```

```
}
```

```
let i = 0;
```

```
{
```

```
{
```

```
if (hasBubbleGum[i])
```

```
while (i > hasBubbleGum.length)
```

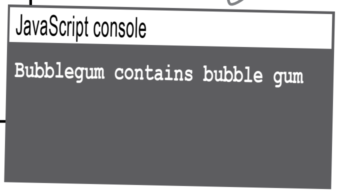
```
let products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
```

```
let hasBubbleGum = [false,
                    false,
                    false,
                    true];
```

```
console.log(products[i] +
     " contains bubble gum");
```

Rearrange the magnets here.

Here's the output we're expecting.



```
Bubblegum contains bubble gum
```

Solution on page 176

But wait, there's a better way to iterate over an array

We should really apologize. We can't believe it's already Chapter 4 and we haven't even introduced you to the **for loop**. Think of the for loop as the while loop's cousin. The two basically do the same thing, except the for loop is usually a little more convenient to use. Check out the while loop we just used, and we'll see how that maps into a for loop.

```
let i = 0;
while (i < scores.length) {
  let output = "Bubble solution #" + i + " score: " + scores[i];
  console.log(output);
  i = i + 1;
}
```

Ⓐ First we INITIALIZED a counter.

Ⓑ Then we tested that counter in a CONDITIONAL expression.

We also had a BODY to execute; that is, all the statements between the { and }.

Ⓒ And finally, we INCREMENTED the counter.

Now let's look at how the for loop makes all that so much easier:

A for loop starts with the keyword for.

Ⓐ In the parentheses, there are three parts. The first part is the loop variable INITIALIZATION. This initialization happens only once, before the for loop starts.

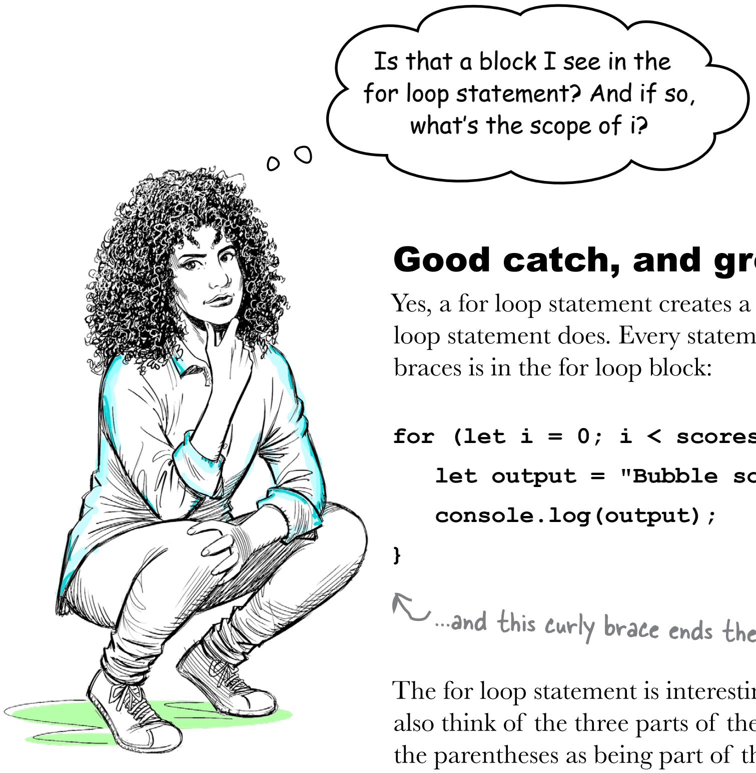
Ⓑ The second part is the CONDITIONAL test. Each time we loop, we perform this test, and if it is false, we stop.

Ⓒ And the third part is where we INCREMENT the counter. This happens once per loop, after all the statements in the BODY.

```
for (let i = 0; i < scores.length; i = i + 1) {
    let output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
}
```

Note the semicolons after the first two parts of the for loop.

The BODY goes here. Notice there are no changes other than moving the increment of i into the for statement.


Is that a block I see in the for loop statement? And if so, what's the scope of i?

Good catch, and great question.

Yes, a for loop statement creates a block just like a while loop statement does. Every statement between the curly braces is in the for loop block:

```
for (let i = 0; i < scores.length; i = i + 1) {
   let output = "Bubble solution #" + i + " score: " + scores[i];
   console.log(output);
}
```

This curly brace starts the block...

Every statement between the curly braces is in the block created by the for loop.

...and this curly brace ends the block.

The for loop statement is interesting because you can also think of the three parts of the statement between the parentheses as being part of the block:

```
for (let i = 0; i < scores.length; i = i + 1) {
   ...
}
```

Think of these pieces of the for loop as being statements in the block created by the for loop.

That means the declaration of the `i` loop variable is local to the block, meaning the value of `i` will not be visible outside the block. So you would **not** be able to do this, for instance:

```
for (let i = 0; i < scores.length; i = i + 1) {
   let output = "Bubble solution #" + i + " score: " + scores[i];
   console.log(output);
}
console.log("There are " + i + " bubble solutions in scores");
```

If you try this at home, you'll see an error in the console: Uncaught ReferenceError: i is not defined.

Sharpen your pencil

Rewrite your fridge magnet code (from three pages back) so that it uses a for loop instead of a while loop. If you need a hint, refer to each piece of the while loop on page 146 and see how it maps to the corresponding location in the for loop.

```
let products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];

    let hasBubbleGum = [false,
                        false,
                        false,
                        true];
   let i = 0;
   while (i < hasBubbleGum.length)      {
        if (hasBubbleGum[i])   {
            console.log(products[i] +
                " contains bubble gum");
        }
        i = i + 1;
   }
```

Your code goes here.

Solution on page 177

We've got all the pieces for the first part of the report; let's put this all together...

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    let scores = [60, 50, 60, 58, 54, 54,
                  58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51,
                  69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18,
                  41, 53, 55, 61, 51, 44];


    for (let i = 0; i < scores.length; i = i + 1) {

        let output = "Bubble solution #" + i +
                     " score: " + scores[i];

        console.log(output);

    }

  </script>
</head>
<body></body>
</html>
```

We've got the standard HTML stuff here for a web page. We don't need much; just enough to create a script.

Here's our bubble scores array.

Here's the for loop we're using to iterate through all the bubble solution scores.

Each time through the loop, we create a string with the value of i, which is the bubble solution number, and scores[i], which is the score that bubble solution got.

(Also notice we split the string up across two lines here. That's okay as long as you don't create a new line in between the quotes that delimit a string. Here, we did it after a concatenation operator (+), so it's okay. Be careful to type it in exactly as you see here.)

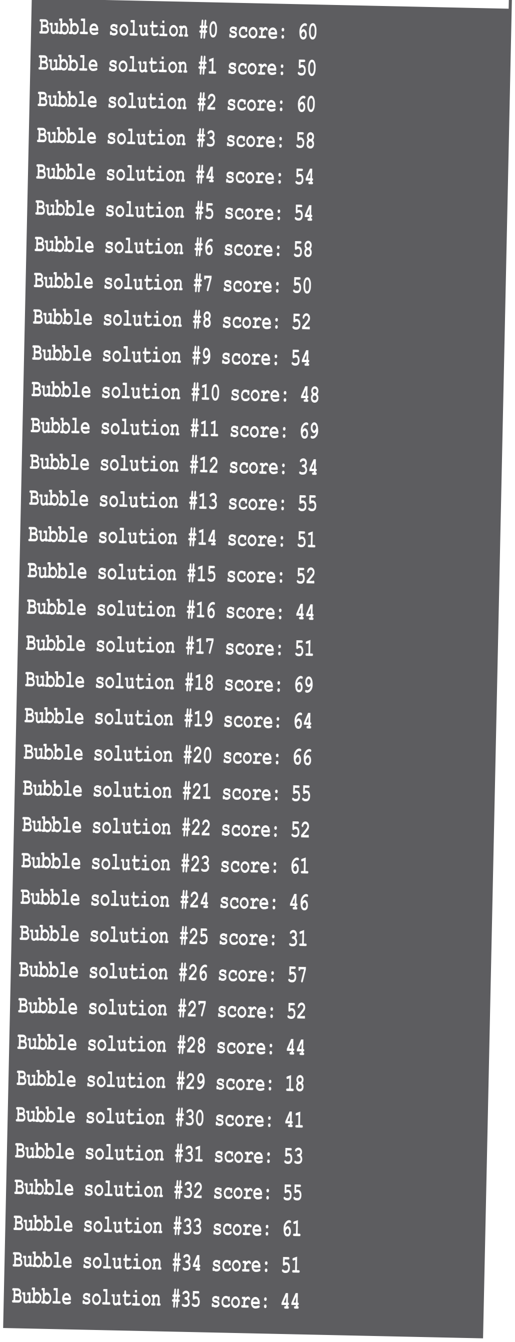
Then we display the string in the console. And that's it! Time to run this report.

Test drive the bubble report

Save this file as "bubbles.html" and load it into your browser. Make sure you've got the console visible (you might need to reload the page if you activate the console after you load the page), and check out the brilliant report you just generated for the Bubbles-R-Us CEO.

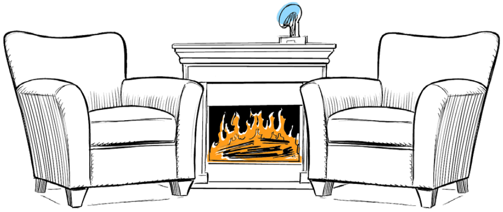
Just what the CEO ordered.

It's nice to see all the bubble scores in a report, but it's still hard to find the highest score. We need to work on the rest of the report requirements to make it a little easier to find the winner.



JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
Bubble solution #3 score: 58
Bubble solution #4 score: 54
Bubble solution #5 score: 54
Bubble solution #6 score: 58
Bubble solution #7 score: 50
Bubble solution #8 score: 52
Bubble solution #9 score: 54
Bubble solution #10 score: 48
Bubble solution #11 score: 69
Bubble solution #12 score: 34
Bubble solution #13 score: 55
Bubble solution #14 score: 51
Bubble solution #15 score: 52
Bubble solution #16 score: 44
Bubble solution #17 score: 51
Bubble solution #18 score: 69
Bubble solution #19 score: 64
Bubble solution #20 score: 66
Bubble solution #21 score: 55
Bubble solution #22 score: 52
Bubble solution #23 score: 61
Bubble solution #24 score: 46
Bubble solution #25 score: 31
Bubble solution #26 score: 57
Bubble solution #27 score: 52
Bubble solution #28 score: 44
Bubble solution #29 score: 18
Bubble solution #30 score: 41
Bubble solution #31 score: 53
Bubble solution #32 score: 55
Bubble solution #33 score: 61
Bubble solution #34 score: 51
Bubble solution #35 score: 44
```



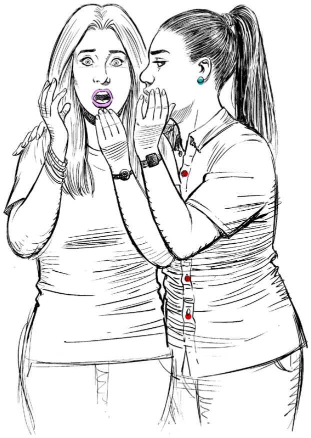
Fireside Chats

Tonight's talk: **The WHILE and FOR loop answer the question "Who's more important?"**

The WHILE loop:	**The FOR loop:**
What, are you kidding me? Hello? I'm the *general* looping construct in JavaScript. I'm not married to looping with a silly counter. I can be used with any type of conditional. Did anyone notice I was taught first in this book?	
	I don't appreciate that tone.
And that's another thing, have you noticed that the FOR loop has no sense of humor? I mean, if we all had to do skull-numbing iteration all day I guess we'd all be that way...	
	Cute. But have you noticed that nine times out of ten, coders use FOR loops?
Oh, I don't think that could possibly be true.	
	Not to mention, doing iteration over, say, an array that has a fixed number of items with a WHILE loop is just a bad, clumsy practice.
This book just showed that FOR and WHILE loops are pretty much the same thing, so how could that be?	
	Ah, so you admit we're more equal than you let on, huh? I'll tell you why...

The WHILE loop:	**The FOR loop:**
	When you use a WHILE loop you have to initialize your counter and increment your counter in separate statements. If, after lots of code changes, you accidentally moved or deleted one of these statements, well, then things could get ugly. But with a FOR loop, everything is packaged right in the FOR statement for all to see and with no chance of things getting changed or lost.
Well, isn’t that nice and neat of you. Hey, most of the iteration I see doesn’t even include counters; it’s stuff like: `while (answer != "forty-two")`	
Try that with a FOR loop!	
	Okay: `for (;answer != "forty-two";)`
Hah, I can’t believe that even works.	
	Oh, it does.
Lipstick on a pig.	
	So that’s all you got? You’re better when you’ve got a general conditional?
Not only better, prettier.	
	Oh, I didn’t realize this was a beauty contest as well.

It's that time again... can we talk about your verbosity?



You've been writing lots of code that looks like this:

Assume myImportantCounter contains a number, like zero.

Here we're taking the variable and incrementing it by one.

```
myImportantCounter = myImportantCounter + 1;
```

After this statement completes, the value of myImportantCounter is one greater than before.

In fact, this kind of statement is so common there's a shortcut for it in JavaScript. It's called the **post-increment** operator, and despite its fancy name, it's quite simple. Using the post-increment operator, we can replace the above line of code with this:

Just add "++" to the variable name.

```
myImportantCounter++;
```

After this statement completes, the value of myImportantCounter is one greater than before.

Of course, it just wouldn't feel right if there wasn't a **post-decrement** operator as well. You can use the post-decrement operator on a variable to reduce its value by one, like this:

Just add "--" to the variable name.

```
myImportantCounter--;
```

After this statement completes, myImportantCounter is one less than before.

And why are we telling you this now? Because these are commonly used with `for` statements. Let's clean up our code a little using the post-increment operator…

Redoing the for loop with the post-increment operator

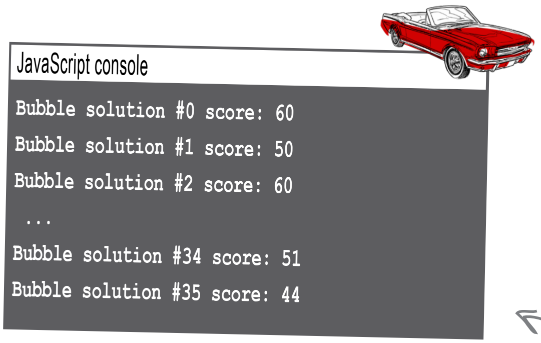
Let's do a quick rewrite and test to make sure the code works the same as before:

```
let scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

for (let i = 0; i < scores.length; i++) {

    let output = "Bubble solution #" + i +
                    " score: " + scores[i];

    console.log(output);

}
```

All we've done is update where we increment the loop variable with the post-increment operator.

Another quick test drive

Time to do a quick test drive to make sure the change to use the post-increment operator works. Save your file, "bubbles.html", and reload. You should see the same report you saw before.

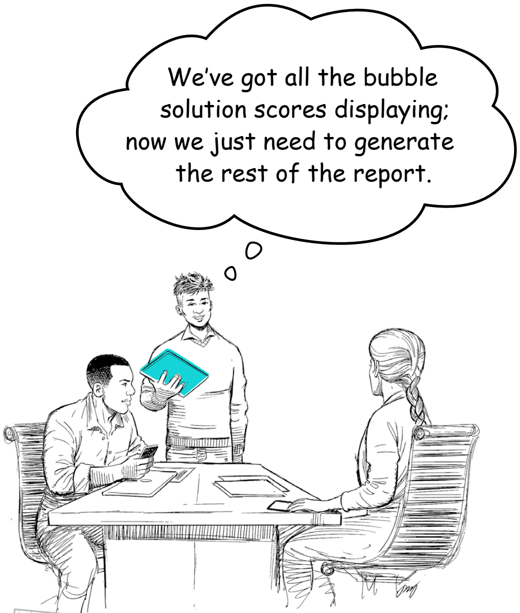


```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
```

The report looks exactly the same.

We're saving a few trees and not showing all the bubble solution scores, but they are all there.

Cubicle Conversation continued...


We've got all the bubble solution scores displaying; now we just need to generate the rest of the report.

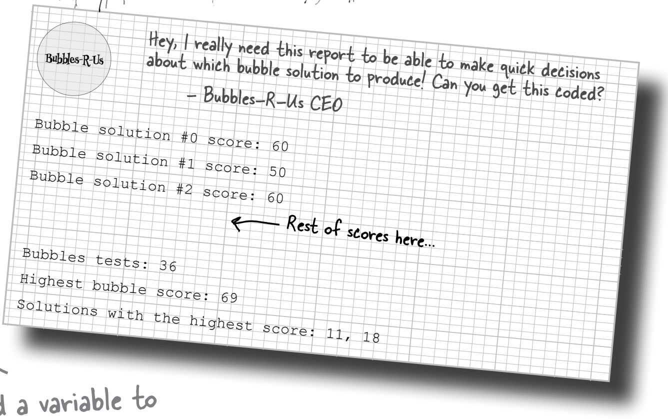
Mara: Right, and the first thing we need to do is determine the total number of bubble tests. That's easy; it's just the length of the scores array.

Alex: Oh, right. We've got to find the highest score too, and then the solutions that have the highest score.

Mara: Yeah, that last one is going to be the toughest. Let's work out finding the highest score first.

Alex: Sounds like a good place to start.

Mara: To do that, I think we just need to maintain a highest score variable that keeps track as we iterate through the array. Here, let me write some pseudocode:



Bubbles-R-Us

Hey, I really need this report to be able to make quick decisions about which bubble solution to produce! Can you get this coded?
– Bubbles-R-Us CEO

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
```

← Rest of scores here...

```
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11, 18
```

```
DECLARE a variable highScore and set it to zero.
FOR: let i=0; i < scores.length; i++
    DISPLAY the bubble solution scores[i]
    IF scores[i] > highScore
        SET highScore = scores[i];
    END IF
END FOR
DISPLAY highScore
```

Add a variable to hold the high score.

Check each time through the loop to see if we have a higher score. If so, that's our new high score.

After the loop, we just display the high score.

Alex: Oh nice; you did it with just a few lines added to our existing code.

Mara: Yup. Each time through the array we look to see if the current score is greater than `highScore`, and if so, that's our new high score. Then, after the loop ends we just display the high score.

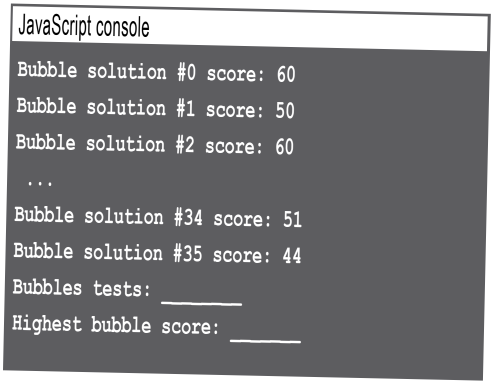
Sharpen your pencil

Go ahead and implement the pseudocode on the previous page to find the highest score by filling in the blanks in the code below. Once you're done, give it a try in the browser by updating the code in "bubbles.html" and reloading the page. Check the results in the console, and fill in the blanks in our console display below with the number of bubble tests and the highest score. Check your answer at the end of the chapter before you go on.

```
let scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

let highScore = _____;
for (let i = 0; i < scores.length; i++) {
    let output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (_____________ > highScore) {
        _____________ = scores[i];
    }
}
console.log("Bubbles tests: " + ____________);
console.log("Highest bubble score: " + _____________);
```

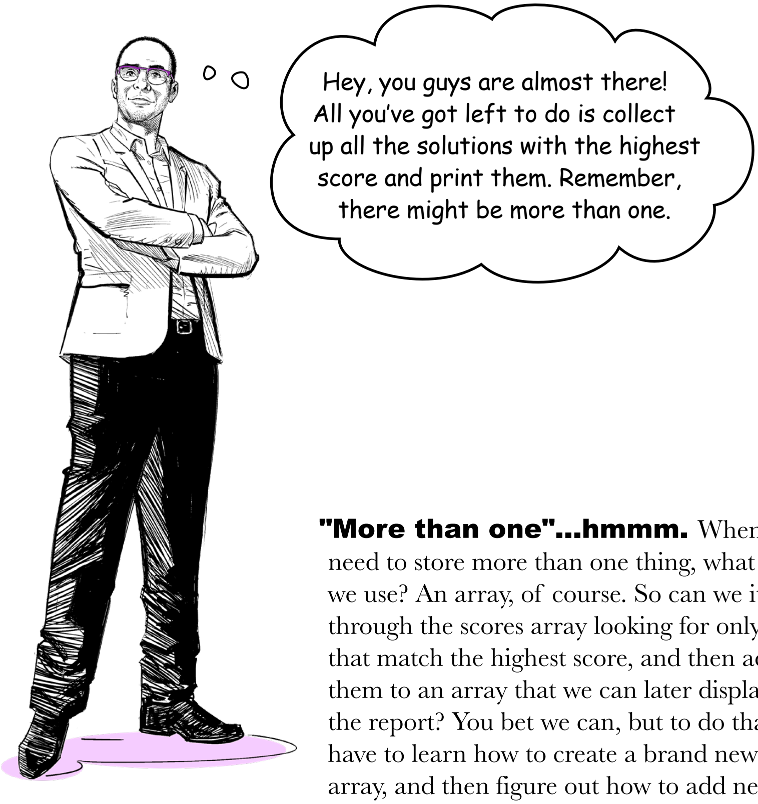
Fill in the blanks to complete the code here...

...and then fill in the blanks showing the output you get in the console.



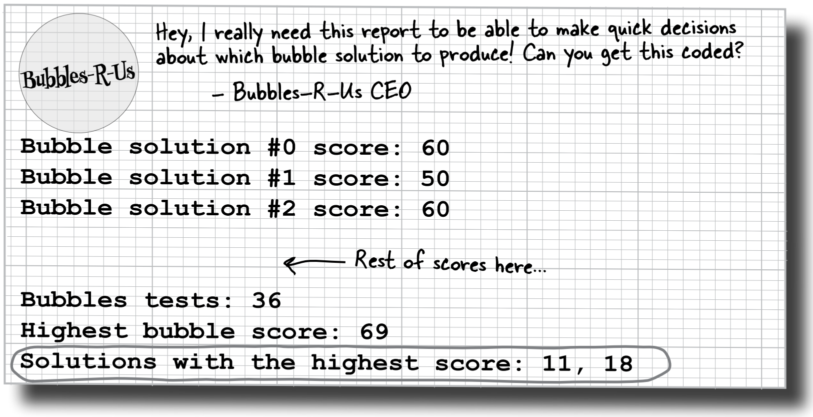
```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: ______
Highest bubble score: _____
```

Solution on page 178


Hey, you guys are almost there! All you've got left to do is collect up all the solutions with the highest score and print them. Remember, there might be more than one.

"More than one"...hmmm. When we need to store more than one thing, what do we use? An array, of course. So can we iterate through the scores array looking for only scores that match the highest score, and then add them to an array that we can later display in the report? You bet we can, but to do that we'll have to learn how to create a brand new empty array, and then figure out how to add new elements to it.

Here's what we have left to do.


Hey, I really need this report to be able to make quick decisions about which bubble solution to produce! Can you get this coded?

– Bubbles-R-Us CEO

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60

Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11, 18
```

Rest of scores here...

Creating an array from scratch (and adding to it)

Before we take on finishing this code, let's get a sense of how to create a new array and add new items to it. You already know how to create an array with values, like this:

```
let genres = ["80s", "90s", "Electronic", "Folk"];
```

Remember, this is called an array literal, because we're literally writing out what goes in the array.

But you can also omit the initial items and just create an empty array:

```
let genres = [];
```

A new array, all ready to go with no items and a length of zero.

This is an array literal too, it just doesn't have anything in it (yet).

And you already know how to add new values to an array. To do that you just assign a value to an item at an index, like this:

```
let genres = [];
genres[0] = "Rockabilly";
genres[1] = "Ambient";
let size = genres.length;
```

A new array item is created that holds the string "Rockabilly".

And a second array item is created that holds the string "Ambient".

And here, size holds the value 2, the length of the array.

When adding new items this way, you have to be careful about which index you're adding. Otherwise, you might create a **sparse** array, which is an array with "holes" in it (like an array with values at 0 and 2, but no value at 1). Having a sparse array isn't necessarily a bad thing, but it does require special attention. There's another way to add new items without worrying about the index, and that's `push`. Here's how it works:

```
let genres = [];
genres.push("Rockabilly");
genres.push("Ambient");
let size = genres.length;
```

Creates a new item in the index at the end of the array (which happens to be 0) and sets its value to "Rockabilly".

Creates another new item at the end of the array (in this case, at index 1) and sets the value to "Ambient".

there are no Dumb Questions

Q: The for statement contains a variable declaration and initialization in the first part of the statement. You said we should put our variable declarations at the top. So, what gives?

A: Yes, putting your variable declarations at the top (of your file, if they are global, or of your function if they are local) is a good practice. However, there are times when it makes sense to declare a variable right where you're going to use it, and a for statement is one of those times. Typically, you use a loop variable, like i, just for iterating, and once the loop is done, you're done with that variable. In that case, just declaring it right in the for statement keeps things tidy. And remember that the scope of i is limited to the block of the for loop statement, so that means you could use i again later on in another for loop if you wanted to.

We're doing the same with the variable output too: we're declaring it inside the for loop block, because we don't need it anywhere else.

Q: What does the syntax myarray.push(value) actually mean?

A: Well, we've been keeping a little secret from you: in JavaScript, an array is actually a special kind of object. As you'll learn in the next chapter, an object can have functions associated with it that act on the object. So, think of push as a function that can act on myarray. In this case, what that function does is add a new item to the array—the item that you pass as an argument to push. So, if you write:

```
genres.push("Metal");
```

you're calling the function push and passing it a string argument, "Metal". The push function takes that argument and adds it as a new value on the end of the genres array.

When you see:

```
myarray.push(value)
```

just think, "I'm pushing a new value onto the end of my array."

Q: Can you say a little more about what a sparse array is?

A: A sparse array is just an array that has values at only a few indices and no values in between. You can create a sparse array easily, like this:

```
let sparseArray = [ ];
sparseArray[0] = true;
spraseArray[100] = true;
```

In this example, the sparseArray has only two values, both true, at indices 0 and 100. The values at all the other indices are undefined. The length of the array is 101 even though there are only two values.

Q: Say I have an array of length 10, and I add a new item at index 10000. What happens with indices 10 through 9999?

A: All those array indices get the value undefined. If you remember, undefined is the value assigned to a variable that you haven't initialized. So, think of this as if you're creating 9,989 variables, but not initializing them. Remember that all those variables take up memory in your computer, even if they don't have a value, so make sure you have a good reason to create a sparse array.

Q: So, if I'm iterating through an array, and some of the values are undefined, should I check to make sure before I use them?

A: If you think your array might be sparse, or even have just one undefined value in it, then yes, you should probably check to make sure that the value at an array index is not undefined before you use it. If all you're doing is displaying the value in the console, then it's no big deal, but it's much more likely that you'll actually want to use that value somehow, perhaps in a calculation of some kind. In that case, if you try to use undefined, you might get an error, or at the very least some unexpected behavior. To check for undefined, just write:

```
if (myarray[i] == undefined) {
  ...
}
```

Notice there are no quotes around undefined (because it's not a string, it's a value).

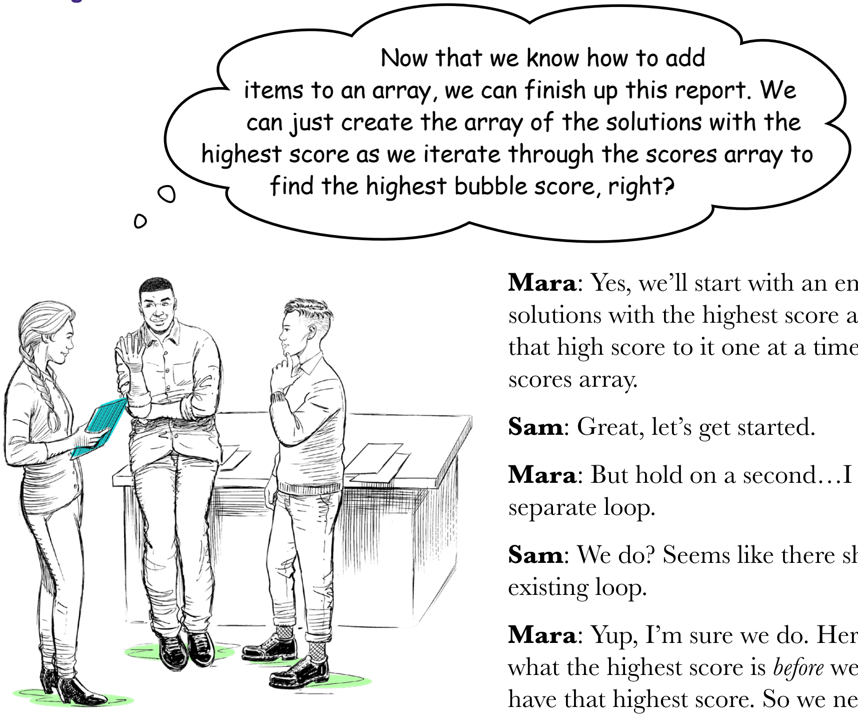
Q: All the arrays we've created so far have been literal. Is there another way to create an array?

A: Yes. You may see this syntax:

```
let myarray = new Array(3);
```

What this does is create a new array with three empty spots in it (that is, an array with length 3, but no values yet). Then you can fill them, just like you normally would, by providing values for myarray at indices 0, 1, and 2. Until you add values yourself, the values in myarray are undefined.

An array created this way is just the same as an array literal, and in practice, you'll find yourself using the literal syntax more often. Don't worry about the details of the syntax above for now (like "new" and why Array is capitalized); we'll get to all that later!


Now that we know how to add items to an array, we can finish up this report. We can just create the array of the solutions with the highest score as we iterate through the scores array to find the highest bubble score, right?

Mara: Yes, we'll start with an empty array to hold the solutions with the highest score and add each solution that has that high score to it one at a time as we iterate through the scores array.

Sam: Great, let's get started.

Mara: But hold on a second…I think we might need a separate loop.

Sam: We do? Seems like there should be a way to do it in our existing loop.

Mara: Yup, I'm sure we do. Here's why: we have to know what the highest score is *before* we can find all the solutions that have that highest score. So we need two loops: one to find the highest score, which we've already written, and then a second one to find all the solutions that have that score.

Sam: Oh, I see. And in the second loop, we'll compare each score to the highest score, and if it matches, we'll add the index of the bubble solution score to the new array we're creating for the solutions with the highest score.

Mara: Exactly! Let's do it.

Sharpen your pencil

Can you write the loop to find all the scores that match the high score? Give it a shot below before you turn the page to see the solution and give it a test drive.

Remember, the variable highScore has the highest score in it; you can use that in the code below.

Here's the new array we'll use to store the bubble solutions with the highest score.

```
let bestSolutions = [];
for (let i = 0; i < scores.length; i++) {



}
```

Your code here.

Solution on page 161

Sharpen your pencil Solution

From page 160

Can you write the loop to find all the scores that match the high score? Here's our solution.

Again, we're starting by creating a new array that will hold all the bubble solutions that match the highest score.

Next, we iterate through the entire scores array, looking for those items with the highest score.

```
let bestSolutions = [];

for (let i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {

        bestSolutions.push(i);

    }
}

console.log("Solutions with the highest score: " + bestSolutions);
```

Each time through the loop, we compare the score at index i with the highScore; if they are equal, then we add that index to the bestSolutions array using push.

And finally, we can display the bubble solutions with the highest score. Notice we're using console.log to display the bestSolutions array. We could create another loop to display the array items one by one, but luckily, console.log will do this for us (and, if you look at the output, it also adds commas between the array values!).

Brain Power

Take a look at the code in the exercise above. What if you woke up and push no longer existed? Could you rewrite this code without using push? Work that code out here:

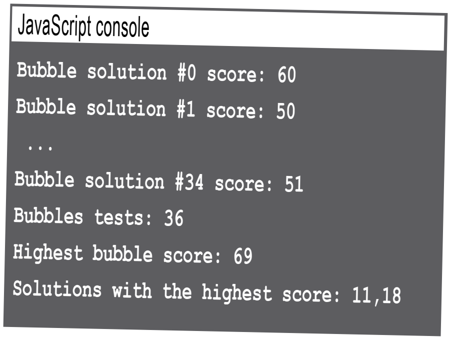
Test drive the final report

Go ahead and add the code to identify the bubble solutions with the highest score to your code in "bubbles.html", then run another test drive. All the JavaScript code is shown below:

```
let scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

let highScore = 0;
for (let i = 0; i < scores.length; i++) {
    let output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (scores[i] > highScore) {
        highScore = scores[i];
    }
}
console.log("Bubbles tests: " + scores.length);
console.log("Highest bubble score: " + highScore);

let bestSolutions = [];
for (let i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {
        bestSolutions.push(i);
    }
}
console.log("Solutions with the highest score: " + bestSolutions);
```

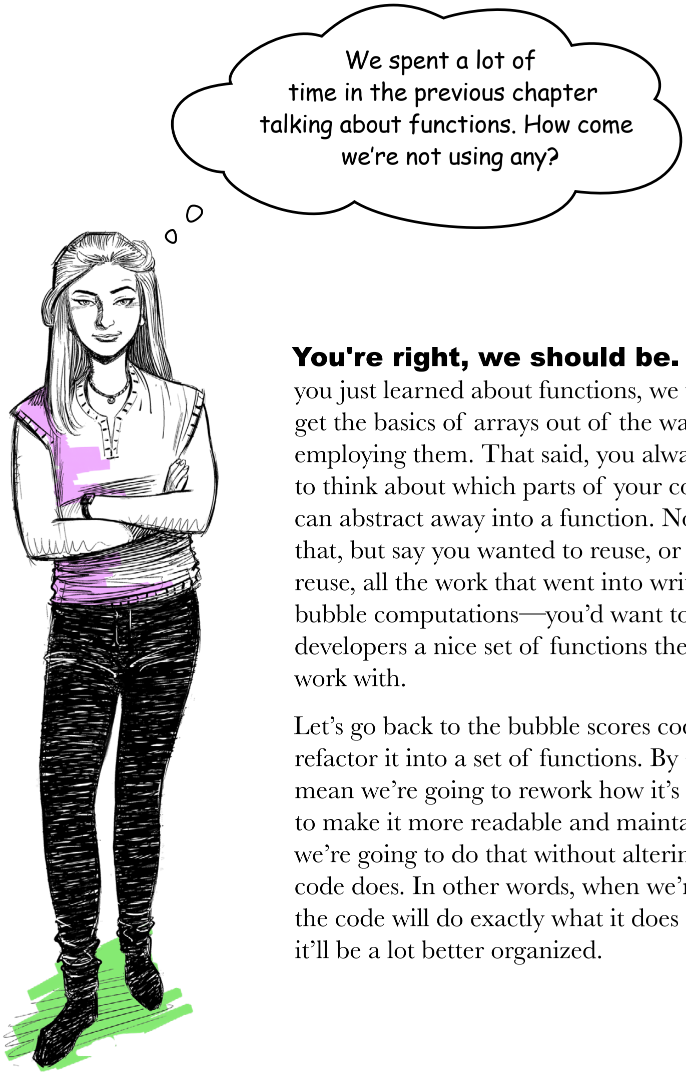
And the winners are...

Bubble solutions #11 and #18 both have a high score of 69! So they are the best bubble solutions in this batch of test solutions.



JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
 ...
Bubble solution #34 score: 51
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11,18
```


We spent a lot of time in the previous chapter talking about functions. How come we're not using any?

You're right, we should be. Given that you just learned about functions, we wanted to get the basics of arrays out of the way before employing them. That said, you always want to think about which parts of your code you can abstract away into a function. Not only that, but say you wanted to reuse, or let others reuse, all the work that went into writing the bubble computations—you'd want to give other developers a nice set of functions they could work with.

Let's go back to the bubble scores code and refactor it into a set of functions. By *refactor*, we mean we're going to rework how it's organized to make it more readable and maintainable, but we're going to do that without altering what the code does. In other words, when we're done, the code will do exactly what it does now, but it'll be a lot better organized.

A quick survey of the code

Let's get an overview of the code we've written and figure out which pieces we want to abstract into functions:

Here's the Bubbles-R-Us code.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    let scores = [60, 50, 60, 58, 54, 54,
                  58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51,
                  69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18,
                  41, 53, 55, 61, 51, 44];

    let highScore = 0;

    for (let i = 0; i < scores.length; i++) {
      let output = "Bubble solution #" + i + " score: " + scores[i];
      console.log(output);
      if (scores[i] > highScore) {
        highScore = scores[i];
      }
    }
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    let bestSolutions = [];

    for (let i = 0; i < scores.length; i++) {
      if (scores[i] == highScore) {
        bestSolutions.push(i);
      }
    }

    console.log("Solutions with the highest score: " + bestSolutions);
  </script>
</head>
<body> </body>
</html>
```

We don't want to declare scores inside the functions that operate on scores because these are going to be different for each use of the functions. Instead, we'll pass the scores as an argument into the functions, so the functions can use any scores array to generate results.

We use this first chunk of code to output each score and at the same time compute the highest score in the array. We could put this in a printAndGetHighScore function.

And we use this second chunk of code to figure out the best results given a high score. We could put this in a getBestResults function.

Writing the printAndGetHighScore function

We've got the code for the `printAndGetHighScore` function already. It's just the code we've already written, but to make it a function, we need to think through what arguments we're passing it and if it returns anything back to us.

Passing in the scores array seems like a good idea, because that way we can reuse the function on other arrays of bubble scores. And we want to return the high score that we compute in the function, so the code that calls the function can do interesting things with it (and, after all, we're going to need it to figure out the best solutions).

Oh, and another thing: often you want your functions to do *one thing* well. Here, we're doing two things: we're displaying all the scores in the array, and we're also computing the high score. We might want to consider breaking this into two functions, but given how simple things are right now we're going to resist the temptation. If we were working in a professional environment we might reconsider and break this into two functions, `printScores` and `getHighScore`. For now, though, we'll stick with one function. Let's get this code refactored:

We've created a function that expects one argument, the scores array.

```
function printAndGetHighScore(scores) {
    let highScore = 0;
    for (let i = 0; i < scores.length; i++) {
        let output = "Bubble solution #" + i + " score: " + scores[i];
        console.log(output);
        if (scores[i] > highScore) {
            highScore = scores[i];
        }
    }
    return highScore;
}
```

This code is exactly the same. Well, actually it LOOKS exactly the same, but it now uses the parameter scores rather than the global variable scores.

And we've added one line here to return the highScore to the code that called the function.

Refactoring the code using printAndGetHighScore

Now, we need to change the rest of the code to use our new function. To do so, we simply call the new function and set the variable `highScore` to the result of the `printAndGetHighScore` function:

```
<!doctype html>
<html lang="en">
<head>
  <title>Bubble Factory Test Lab</title>
  <meta charset="utf-8">
  <script>
    let scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

    function printAndGetHighScore(scores) {
        let highScore = 0;
        for (let i = 0; i < scores.length; i++) {
            let output = "Bubble solution #" + i + " score: " + scores[i];
            console.log(output);
            if (scores[i] > highScore) {
                highScore = scores[i];
            }
        }
        return highScore;
    }

    let highScore = printAndGetHighScore(scores);
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    let bestSolutions = [];

    for (let i = 0; i < scores.length; i++) {
      if (scores[i] == highScore) {
        bestSolutions.push(i);
      }
    }

    console.log("Solutions with the highest score: " + bestSolutions);
  </script>
</head>
<body> </body>
</html>
```

Here's our new function, all ready to use.

And now we just call the function, passing in the scores array. We assign the value it returns to the variable highScore.

Now we need to refactor this code into a function and make the appropriate changes to the rest of the code.

Sharpen your pencil

Let’s work through this next one together. The goal is to write a function to create an array of bubble solutions that have the high score (there might be more than one, so that's why we're using an array). We’re going to pass this function the scores array and the highScore we computed with printAndGetHighScore. Finish the code below. You’ll find the answer on the next page, but don’t peek! Do the code yourself first, so you really get it.

Here’s the original code in case you need to refer to it.

```
let bestSolutions = [];
for (let i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {
        bestSolutions.push(i);
    }
}
console.log("Solutions with the highest score: " + bestSolutions);
```

We’ve already started this, but we need your help to finish it!

```
function getBestResults(__________, ____________) {
    let bestSolutions = ______;
    for (let i = 0; i < scores.length; i++) {
        if (___________ == highScore) {
            bestSolutions.__________;
        }
    }
    return _______________;
}

let bestSolutions = _____________(scores, highScore);
console.log("Solutions with the highest score: " + bestSolutions);
```

Solution on page 168

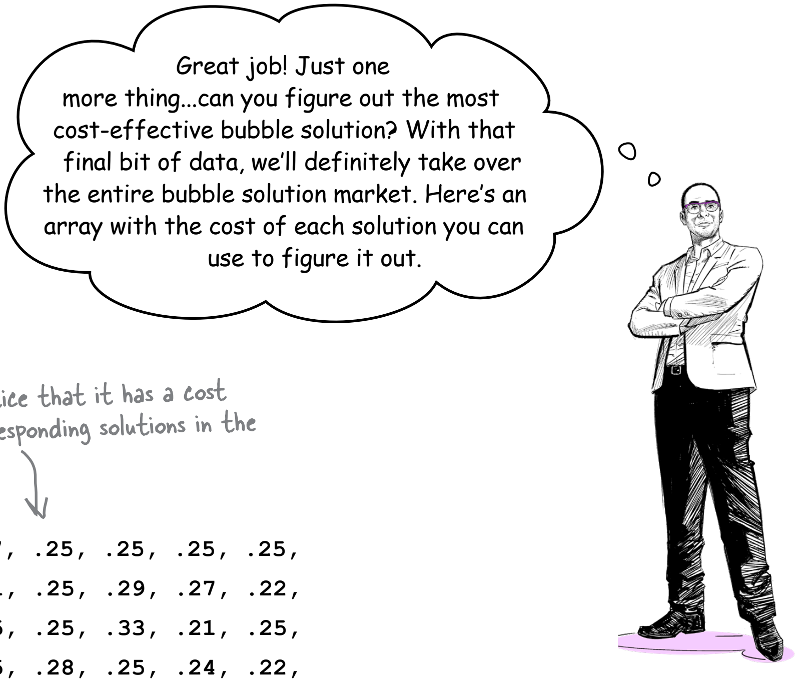
Putting it all together

Once you've completed refactoring your code, make all the changes to "bubbles.html", just like we have below, and reload the bubble report. You should get exactly the same results as before, but now you know your code is more organized and reusable. Create your own scores array and try some reuse!

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    let scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

    function printAndGetHighScore(scores) {
        let highScore = 0;
        for (let i = 0; i < scores.length; i++) {
            let output = "Bubble solution #" + i + " score: " + scores[i];
            console.log(output);
            if (scores[i] > highScore) {
                highScore = scores[i];
            }
        }
        return highScore;
    }

    function getBestResults(scores, highScore) {
        let bestSolutions = [];
        for (let i = 0; i < scores.length; i++) {
            if (scores[i] == highScore) {
                bestSolutions.push(i);
            }
        }
        return bestSolutions;
    }

    let highScore = printAndGetHighScore(scores);
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    let bestSolutions = getBestResults(scores, highScore);
    console.log("Solutions with the highest score: " + bestSolutions);

  </script>
</head>
<body> </body>
</html>
```

Okay, here's the new getBestResults function.

And we use the result of that function to display the best solutions in the report.


Great job! Just one more thing...can you figure out the most cost-effective bubble solution? With that final bit of data, we'll definitely take over the entire bubble solution market. Here's an array with the cost of each solution you can use to figure it out.

Here's the array. Notice that it has a cost for each of the corresponding solutions in the scores array.

```
let costs = [.25, .27, .25, .25, .25, .25,
             .33, .31, .25, .29, .27, .22,
             .31, .25, .25, .33, .21, .25,
             .25, .25, .28, .25, .24, .22,
             .20, .25, .30, .25, .24, .25,
             .25, .25, .27, .25, .26, .29];
```

So, what's the job here? It's to take the leading bubble solutions—that is, the ones with the highest bubble scores—and choose the lowest-cost one. Now, luckily, we've been given a `costs` array that mirrors the `scores` array. That is, the bubble solution score at index 0 in the `scores` array has the cost at index 0 in the `costs` array (.25), the bubble solution at index 1 in the `scores` array has the cost at index 1 in the `costs` array (.27), and so on. So, for any score you'll find its cost in the `costs` array at the same index. Sometimes we call these *parallel* arrays:

scores and costs are parallel arrays because for each score there is a corresponding cost at the same index.

```
let costs = [.25, .27, .25, .25, .25, .25, .33, .31, .25, .29, .27, .22, ..., .29];
```

The cost at 0 is the cost of the bubble solution at 0...

...and likewise for the other cost and score values in the arrays.

```
let scores = [60,  50,  60,  58,  54,  54,  58,  50,  52,  54,  48,  69,  ..., 44];
```

This seems a little tricky. How do we not only determine the scores that are highest, but then pick the one with the lowest cost?

Mara: Well, we know the highest score already.

Sam: Right, but how do we use that? And we have these two arrays; how do we get those to work together?

Mara: I'm pretty sure either of us could write a simple for loop that goes through the `scores` array again and picks up the items that match the highest score.

Sam: Yeah, I could do that. But then what?

Mara: Anytime we hit a score that matches the highest score, we need to see if its cost is the lowest we've seen.

Sam: Oh I see, so we'll have a variable that keeps track of the index of the lowest-cost high score. Wow, that's a mouthful.

Mara: Exactly. And once we get through the entire array, whatever index is in that variable is the index of the item that not only matches the highest score, but has the lowest cost.

Sam: What if two items match in cost?

Mara: Hmm, we have to decide how to handle that. I'd say, whatever one we see first is the winner. Of course, we could do something more complex, but let's stick with that unless the CEO says differently.

Sam: This is complicated enough that I think I want to sketch out some pseudocode before writing anything.

Mara: I agree; whenever you are managing indices of multiple arrays things can get tricky. Let's do that. In the long run I'm sure it will be faster to plan it first.

Sam: Okay, I'll take a first stab at it…

Exercise

Check out the psuedocode below. Translate it into JavaScript. Make sure to check your answer at the end of the chapter.

```
FUNCTION getMostCostEffectiveSolution (scores, costs, highScore)
DECLARE a variable cost and set it to 100.
DECLARE a variable index.
FOR: let i=0; i < scores.length; i++
    IF the bubble solution at scores[i] has the highest score
        IF the current value of cost is greater than the cost of the bubble solution
        THEN
            SET the value of index to the value of i
            SET the value of cost to the cost of the bubble solution
        END IF
    END IF
END FOR
RETURN index
```

```
function getMostCostEffectiveSolution(scores, costs, highScore) {









}
let mostCostEffective = getMostCostEffectiveSolution(scores, costs, highScore);
console.log("Bubble Solution #" + mostCostEffective + " is the most cost effective");
```

← Translate the pseudocode to JavaScript here.

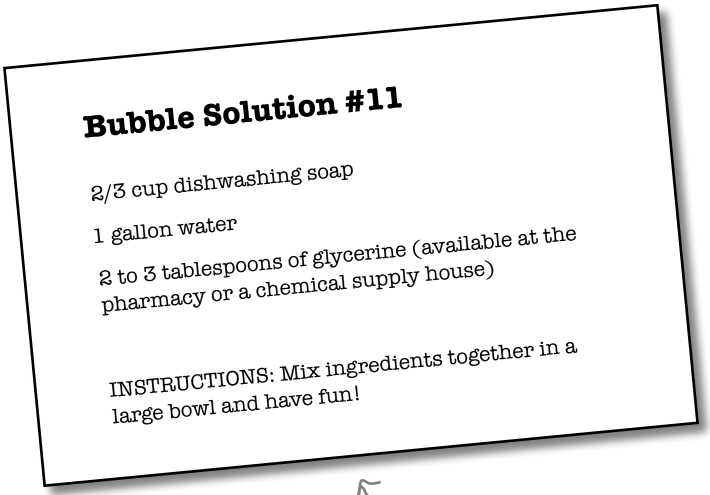
→ Solution on page 179

THE WINNER: SOLUTION #11

The last bit of code you wrote really helped determine the TRUE winner; that is, the solution that produces the most bubbles at the lowest cost. Congrats on taking a lot of data and crunching it down to something Bubbles-R-Us can make real business decisions with.

Now, if you're like us, you're dying to know what is in bubble solution #11. Look no further; the Bubbles-R-Us CEO said he'd be delighted to give you the recipe after all your unpaid work.

So, you'll find the recipe for bubble solution #11 below. Take some time to let your brain process arrays by making a batch, getting out, and blowing some bubbles before you begin the next chapter. Oh, but don't forget the bullet points and the crossword before you go!



Bubble Solution #11

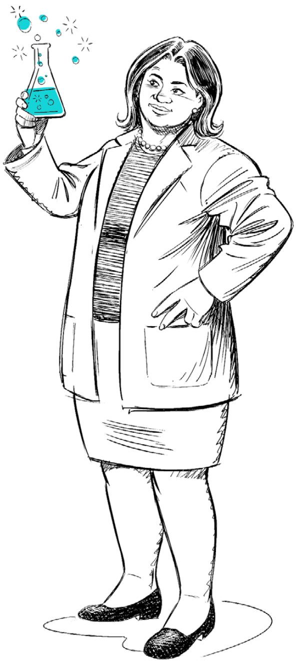
2/3 cup dishwashing soap

1 gallon water

2 to 3 tablespoons of glycerine (available at the pharmacy or a chemical supply house)

INSTRUCTIONS: Mix ingredients together in a large bowl and have fun!

DO try this at home!

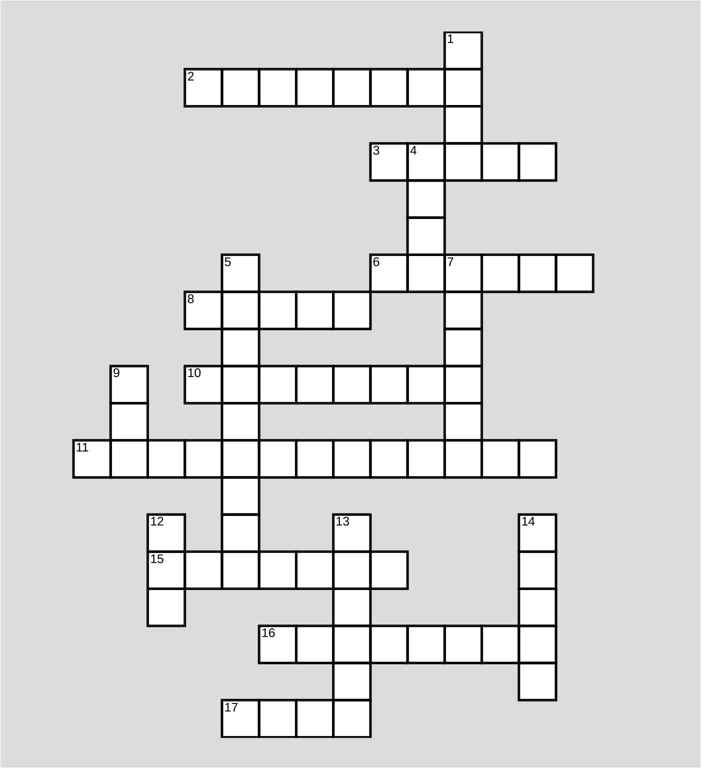


Bullet Points

- Arrays are a **data structure** for ordered data.
- An array holds a set of items, each with its own **index**.
- Arrays use a zero-based index, where the first item is at index zero.
- All arrays have a **length** property, which holds a number representing the number of items in the array.
- You can access any item using its index. For example, use myArray[1] to access item one (the second item in the array).
- If an item doesn't exist, trying to access it will result in a value of undefined.
- Assigning a value to an existing item will change its value.
- Assigning a value to an item that doesn't exist in the array will create a new item in the array.
- You can use a value of any type for an array item.
- Not all the values in an array need to be the same type.
- Use the **array literal notation** to create a new array.
- You can create an empty array with:
  let myArray = [ ];
- The **for loop** is commonly used to iterate through arrays.
- A for loop packages up variable initialization, a conditional test, and variable increment into one statement.
- Like a while loop and an if/else statement, a for loop creates a **block** of code. The **scope** of the loop increment variable, often named i, is the block of the for loop.
- The while loop is most often used when you don't know how many times you need to loop, and you're looping until a condition is met. The for loop is most often used when you know the number of times the loop needs to execute.
- Sparse arrays occur when there are undefined items in the middle of an array.
- You can increment a variable by one with the **post-increment** operator, ++.
- You can decrement a variable by one with the **post-decrement** operator, --.
- You can add a new value to the end of an array using **push**.

JavaScript cross

Let arrays sink into your brain as you do the crossword.



ACROSS

2. Arrays are good for storing _______ values.

3. A for loop creates a _______ scope for the loop variable.

6. To change a value in an array, simply _______ the item a new value.

8. Access an array item using its _______ in square brackets.

10. When you _______ your code, you organize it so it's easier to read and maintain.

11. The operator we use to increment a loop variable.

15. An array is an _______ data structure.

16. Functions can help _______ your code.

17. To add a new value to the end of an existing array, use _______.

DOWN

1. The index of the first item in an array is _______.

4. The last index of an array is always one _______ than the length of the array.

5. The value an array item gets if you don't specify one.

7. An array with undefined values is called a _______ array.

9. How many bubble solutions had the highest score?

12. We usually use a _______ loop to iterate over an array.

13. When iterating through an array, we usually use the _______ property to know when to stop.

14. Each value in an array is stored at an _______.

→ Solution on page 180

From page 133

Look how easy it is to create arrays Solution

You don't know much about arrays yet, but we bet you can make some good guesses about how they work. Take a look at each line of code below and see if you can guess what it does. Here's our solution.

```
let testResults = [91, 84, 100];
let dogs = ["Fido", "Spot"];
let isBarking = [true, false];

let dogBarking1 = isBarking[0];

if (dogBarking1 == true) {
    console.log(dogs[0] + " is barking");
}

if (testResults[2] > 90) {
    console.log("You got an A!");
}
```

Create an array of integer test results.

Create an array of string dog names.

Create an array of booleans.

Get the first value in the isBarking array and store it in the variable dogBarking1.

If dogBarking1 is true,

then print the first dog name in the dogs array to the console.

If the third item in testResults > 90,

then print the string "You got an A!" to the console.

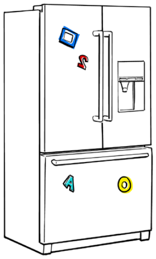
From page 136

Sharpen your pencil Solution

The products array below holds the Jenn and Berry's ice cream flavors. The ice cream flavors were added to this array in the order of their creation. Finish the code to determine the *most recent* ice cream flavor they created. Here's our solution.

```
let products = ["Choo Choo Chocolate", "Icy Mint", "Cake Batter", "Bubblegum"];
let last = products.length - 1;
let recent = products[last];
```

We can use the length of the array minus 1 to get the index of the last item. The length is 4, and the index of the last item is 3, because we start from 0.



Code Magnets Solution

From page 145

We've got code for testing to see which ice cream flavors have bubblegum pieces in them. We had all the code nicely laid out on our fridge using fridge magnets, but the magnets fell on the floor. It's your job to put them back together. Be careful; a few extra magnets got mixed in. Here's our solution.

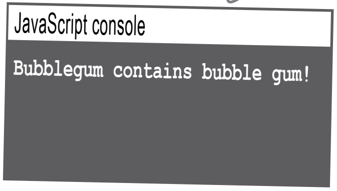
```
let products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
    let hasBubbleGum = [false,
                        false,
                        false,
                        true];
let i = 0;
while (i < hasBubbleGum.length)  {
    if (hasBubbleGum[i])        {
        console.log(products[i] +
            " contains bubble gum");
    }
    i = i + 1;
}
```

Rearrange the magnets here.

Leftover magnets.

```
{      i = i + 2;
while (i > hasBubbleGum.length)
```

Here's the output we're expecting.



JavaScript console

```
Bubblegum contains bubble gum!
```

Sharpen your pencil Solution

From page 148

Rewrite your fridge magnet code (from the previous page) so that it uses a for loop instead of a while loop. If you need a hint, refer to each piece of the while loop on page 146 and see how it maps to the corresponding location in the for loop. Here's our solution.

```
let products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
          let hasBubbleGum = [false,
                              false,
                              false,
                              true];
        let i = 0;
        while (i < hasBubbleGum.length)          {
            if (hasBubbleGum[i])   {
                console.log(products[i] +
                    " contains bubble gum");
            }
            i = i + 1;
        }
```

Your code goes here.

```
let products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
let hasBubbleGum = [false,
                    false,
                    false,
                    true];
for (let i = 0; i < hasBubbleGum.length; i = i + 1) {
    if (hasBubbleGum[i]) {
        console.log(products[i] + " contains bubble gum");
    }
}
```

Sharpen your pencil Solution

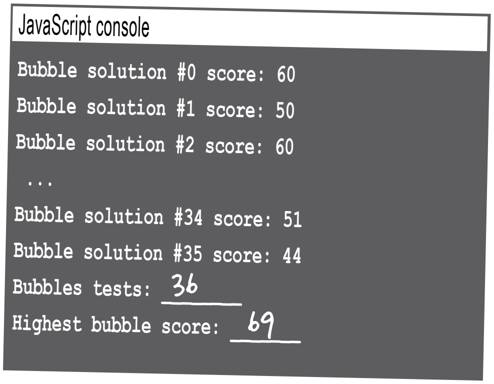
From page 156

Go ahead and implement the pseudocode on page 155 to find the highest score by filling in the blanks in the code below. Once you're done, give it a try in the browser by updating the code in "bubbles.html", and reloading the page. Check the results in the console, and fill in the blanks in our console display below with the number of bubble tests and the highest score. Here's our solution.

```
let scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

let highScore = 0;

for (let i = 0; i < scores.length; i++) {
    let output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (scores[i] > highScore) {
        highScore = scores[i];
    }
}
console.log("Bubbles tests: " + scores.length);
console.log("Highest bubble score: " + highScore);
```

Fill in the blanks to complete the code here...

...and then fill in the blanks showing the output you get in the console.



JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: 36
Highest bubble score: 69
```

Exercise Solution

From page 171

Here's our solution for the getMostCostEffectiveSolution function, which takes an array of scores, an array of costs, and a high score, and finds the index of the bubble solution with the highest score and lowest cost. Go ahead and test drive all your code in "bubbles.html" and make sure you see the same results.

getMostCostEffectiveSolution takes the array of scores, the array of costs, and the high score.

```
function getMostCostEffectiveSolution(scores, costs, highScore) {
    let cost = 100;
    let index;

    for (let i = 0; i < scores.length; i++) {
        if (scores[i] == highScore) {
            if (cost > costs[i]) {
                index = i;
                cost = costs[i];
            }
        }
    }
    return index;
}
let mostCostEffective = getMostCostEffectiveSolution(scores, costs, highScore);
console.log("Bubble solution #" + mostCostEffective + " is the most cost effective");
```

We'll keep track of the lowest-cost solution in cost...

We start cost at a high number, and we'll lower it each time we find a lowercost solution with the high score.

...and the index of the lowest-cost solution in index.

We iterate through the scores array like before...

...and check to see if the solution has the high score.

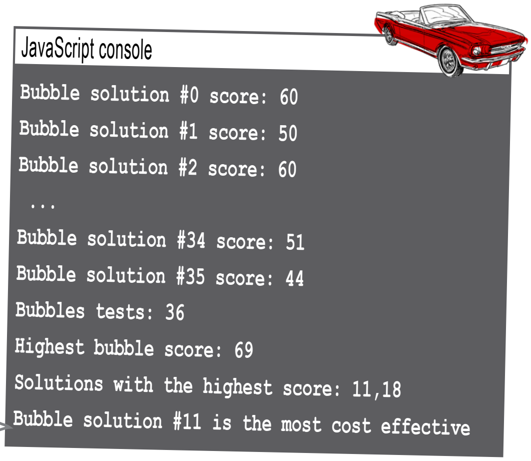
If it does, then we can check its cost. If the current cost is greater than the solution's cost, then we've found a lower-cost solution, so we'll make sure we keep track of which solution it is (its index in the array) and store its cost in the cost variable as the lowest cost we've seen so far.

Once the loop is complete, the index of the solution with the lowest cost is stored in index, so we return that to the code that called the function.

We then display the index (which is the bubble solution #) in the console.

The final report showing bubble solution #11 as the winner of the bubble tests with the highest bubble factor at the lowest cost.

BONUS: We could also implement this using the bestSolutions array so we wouldn't have to iterate through all the scores again. Remember, the bestSolutions array has the indices of the solutions with the highest score. So in that code, we'd use the items in the bestSolutions array to index into the costs array to compare the costs. The resulting code is a little more efficient than this version, but it's also a little bit more difficult to read and understand! If you're interested, we've included the code in the book code download at wickedlysmart.com.



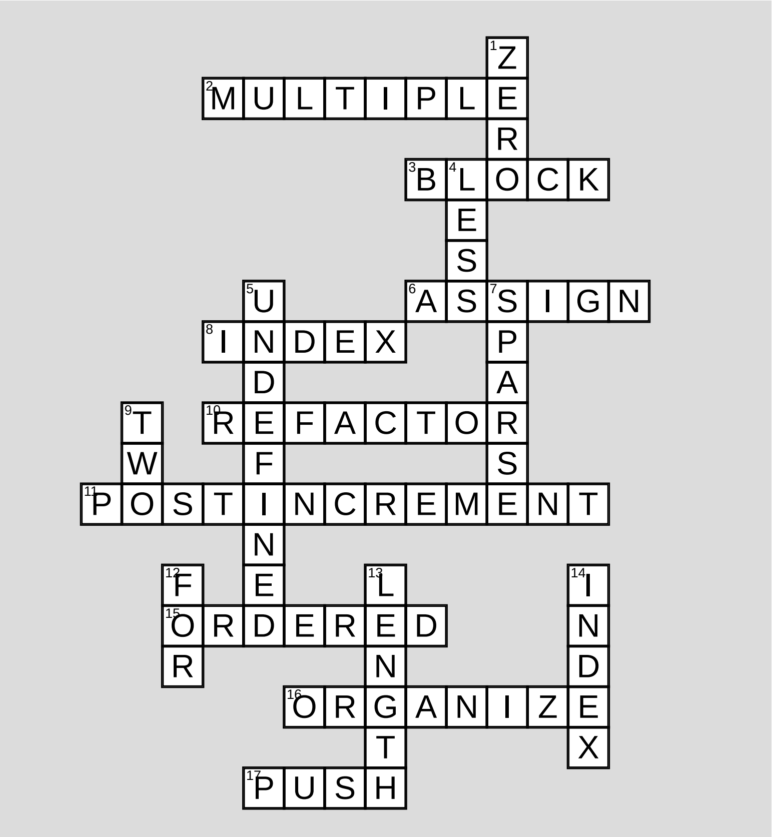
JavaScript console

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11,18
Bubble solution #11 is the most cost effective
```

JavaScript cross Solution

From page 174

Let arrays sink into your brain as you do the crossword.
Here's our solution.


1 ZERO
2 MULTIPLE
3 BLOCK
4 LESS
5 UNDEFINED
6 ASSIGN
7 SPARSE
8 INDEX
9 TWO
10 REFACTOR
11 POSTINCREMENT
12 FOR
13 LENGTH
14 INDEX
15 ORDERED
16 ORGANIZE
17 PUSH

5 understanding objects

A trip to Objectville

So far, you've been using primitives and arrays in your code. And you've approached coding in quite a **procedural manner**, using simple statements, conditionals, and for/while loops with functions—that's not exactly **object-oriented**. In fact, it's not object-oriented *at all!* You did use a few objects here and there without really knowing it, but you haven't written any of your own objects yet. Well, the time has come to leave this boring procedural town behind and create some **objects** of your own. In this chapter, you're going to find out why using objects is going to make your life so much better—well, better in a **programming sense** (we can't really help you with your fashion sense *and* your JavaScript skills all in one book). Just a warning: once you've discovered objects, you'll never want to come back. Send us a postcard when you get there.

Did someone say "objects"?!

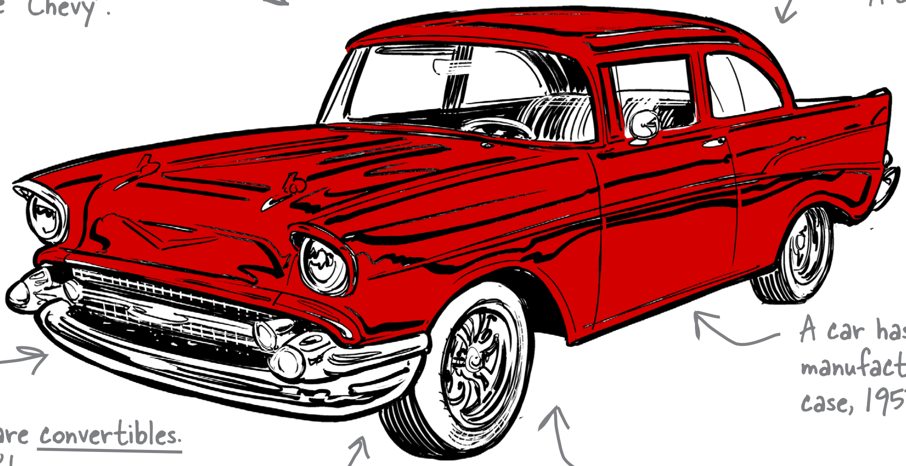
Ah, our favorite topic! Objects are going to take your JavaScript programming skills to the next level—they're the key to managing complex code, to understanding the browser's document model (which we'll get to in the next chapter), to organizing your data...and they're even the fundamental way many JavaScript libraries are packaged up. So objects must be a difficult topic, right? Hah! We're going to jump in head first, and you'll be using them in no time.

Here's the secret to JavaScript objects: they're just collections of properties. Let's take an example—say, a car. A car has various properties:

And a model. In this case we'd call this car a "Bel Air".

Cars have a make, like "Chevy".

A car has a color.



A car has a year of manufacture; in this case, 1957.

Some cars are convertibles. This one isn't.

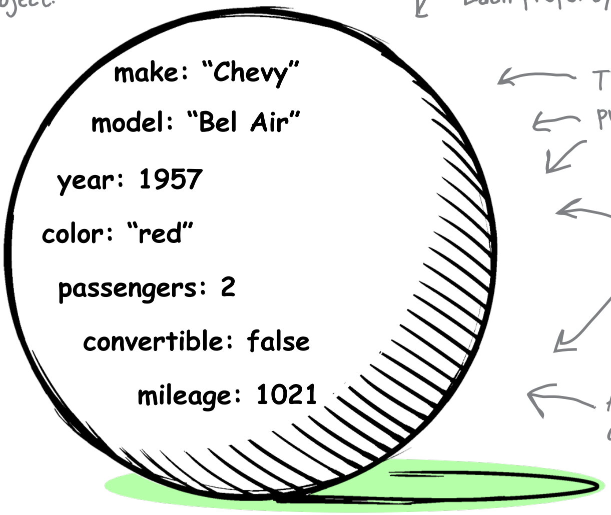
Cars can hold a maximum number of passengers.

Cars have a mileage number that represents how many miles they've been driven.

Cars don't just have properties; they do things too. We'll talk about the behavior of objects a little bit later. For now, let's get back to properties...

Thinking about properties...

Of course, there's a lot more to a real car than just a few properties, but for the purposes of coding, these are the properties we want to capture in software. Let's think about these properties in terms of JavaScript data types:



Here's our car represented as a software object.

The car has a set of properties.

Each property has a name and a value.

The make, model, and color properties have string values.

The year, passengers, and mileage properties are numbers.

And the property convertible is a boolean.

Car Object

Brain Power

Are there other properties you'd want to have in a car object? Go ahead and think through all the properties you might come up with for a car and write them below. Remember, only some real-world properties are going to be useful in software.

Those fuzzy dice may look nice, but would they really be useful in an object?

Sharpen your pencil

We've started making a table of property names and values for a car. Can you help complete it? Make sure you compare your answers with ours before moving on!

Put your property names here.

And put the corresponding values over here.

{
make : "Chevy",
model : ,
year : ,
color : ,
passengers : ,
convertible : ,
mileage : ,
: ,
:
};

Put your answers here. Feel free to expand the list to include your own properties.

When you're done, notice the syntax we've placed around the properties and values. There might be a pop quiz at some point...just sayin'.

Solution on page 228

Brain Power

What if the car is a taxi? What properties and values would it share with your '57 Chevy? How might they differ? What additional properties might it have (or not have)?

How to create an object

Here's the good news: after the last Sharpen your Pencil exercise, you're already most of the way to creating an object. All you really need to do is assign what you wrote on the previous page to a variable (so you can do things with your object after you've created it). Like this:

Add a variable declaration for the object.

Next, start an object with a left curly brace.

Then all the object's properties go inside.

Each property has a name, a colon, and then a value. Here we have strings, numbers, and one boolean as property values.

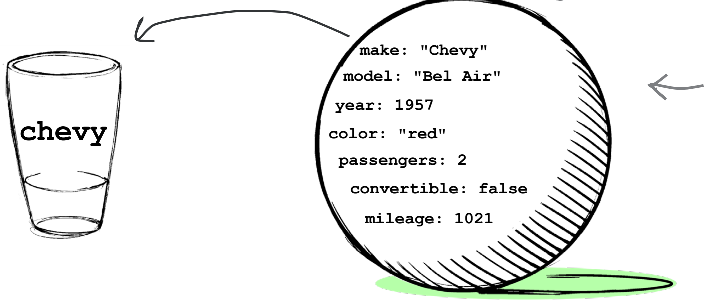
Notice that each property is separated by a comma.

```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

We end the object with a closing curly brace, and just like any other variable declaration, we end this one with a semicolon.

The result of all this? A brand new object, of course. Think of the object as something that holds all your names and values (in other words, your properties) together.

Now you've got a live object complete with a set of properties. And you've assigned your object to a variable that you can use to access and change its properties.



We've taken the textual description of the object above and created a real live JavaScript object from it.

You can now take your object, pass it around, get values from it, change it, add properties to it, or take them away. We'll get to how to do all that in a second. For now, let's create some more objects to play with…

Exercise

You don't have to be stuck with just one object. The real power of objects (as you'll see soon enough) is having lots of objects and writing code that can operate on whatever object you give it. Try your hand at creating another object from scratch…another car object. Go ahead and work out the code for your second object. Check your answer at the end of the chapter before you go on.

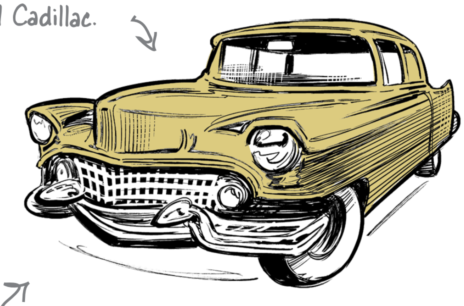
```
let cadi = {









};
```

Put the properties for your Cadillac object here.

This car is a tan color.

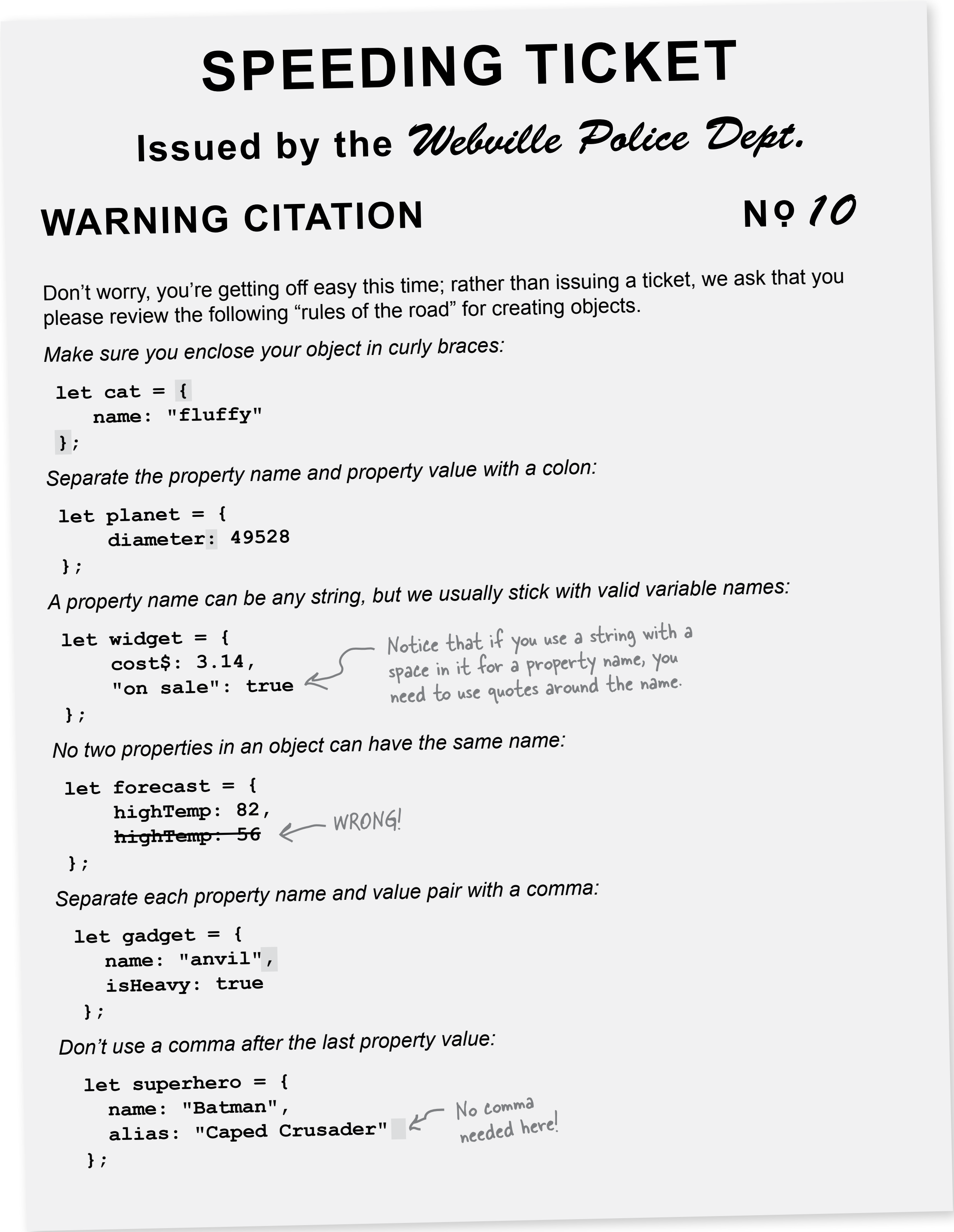
This is a 1955 GM Cadillac.

It's not a convertible, and it can hold five passengers (it's got a nice big bucket seat in the back).



Its mileage is 12,892.

Solution on page 229



SPEEDING TICKET

Issued by the *Webville Police Dept.*

WARNING CITATION

Nº *10*

Don't worry, you're getting off easy this time; rather than issuing a ticket, we ask that you please review the following "rules of the road" for creating objects.

Make sure you enclose your object in curly braces:

```
let cat = {
   name: "fluffy"
};
```

Separate the property name and property value with a colon:

```
let planet = {
    diameter: 49528
};
```

A property name can be any string, but we usually stick with valid variable names:

```
let widget = {
    cost$: 3.14,
    "on sale": true
};
```

Notice that if you use a string with a space in it for a property name, you need to use quotes around the name.

No two properties in an object can have the same name:

```
let forecast = {
    highTemp: 82,
    ~~highTemp: 56~~
};
```

WRONG!

Separate each property name and value pair with a comma:

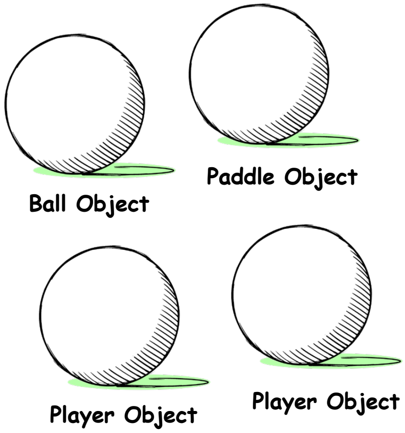
```
let gadget = {
   name: "anvil",
   isHeavy: true
};
```

Don't use a comma after the last property value:

```
let superhero = {
   name: "Batman",
   alias: "Caped Crusader"
};
```

No comma needed here!

What is "object-oriented" anyway?



Up until now, we've been thinking of a problem as a set of variable declarations, conditionals, for/while statements, and function calls. That's thinking *procedurally*: first do this, then do this, and so on. With *object-oriented* programming, we think about a problem in terms of objects. Objects that have state (like a car might have oil and fuel levels) and behavior (like a car can be started, driven, braked, and stopped).

What's the point? Well, object-oriented programming (or OOP for short) allows you to free your mind to think at a higher level. It's the difference between having to toast your bread from first principles (create a heating coil out of wire, hook it to electricity, turn the electricity on and then hold your bread close enough to toast it, not to mention watch long enough for it to toast and then unhook the heating coil), and just using a toaster (place bread in toaster and push down on the toast button). The first way is procedural, while the second way is object-oriented: you have a toaster object that supports an easy method of inserting bread and toasting it.

What do you like about OOP?

"It helps me design in a more natural way. Things have a way of evolving."
-Joy, 27, software architect

"Not messing around with code I've already tested, just to add a new feature."
-Brad, 32, programmer

"I like that the data and the methods that operate on that data are together in one object."
-Josh, 22, beer drinker

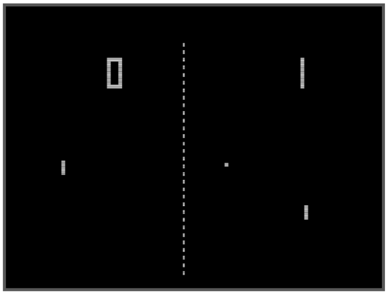
"Reusing code in other apps. When I write a new object, I can make it flexible enough to be used in something new, later."
-Chris, 39, project manager

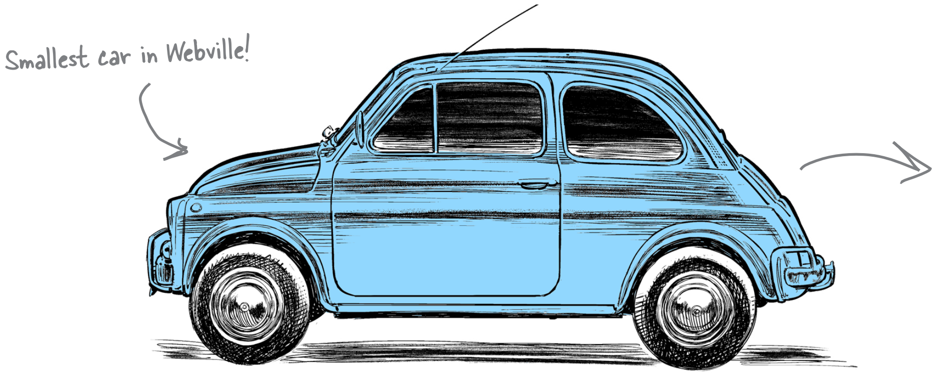
"I can't believe Chris just said that. He hasn't written a line of code in five years."
-Daryl, 44, works for Chris

Brain Power

Say you were implementing a classic Ping-Pong style video arcade game. What would you choose as objects? What state and behavior do you think they'd have?

Pong!





Smallest car in Webville!

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};
```

How properties work

So you've got all your properties packaged up in an object. Now what? Well, you can examine the values of those properties, change them, add new properties, take away properties, and in general, compute using them. Let's try a few of these things out, using JavaScript of course.

How to access a property. To access a property of an object, start with the object name, follow it with a period (otherwise known as a "dot"), and then use the property name. We often call that "dot" notation, and it looks like this:

The "dot" is just a period.

Use the name of the object first...

...then a "dot"...

...then the name of the property.

```
fiat.mileage
```

And then we can use a property in any expression, like this:

```
let miles = fiat.mileage;
if (miles < 2000) {
    buyIt();
}
```

Start with the variable that holds your object and add a period (otherwise known as a dot) and then your property name.

Dot Notation

- Dot notation (.) gives you access to an object's properties.
- For example, `fiat.color` is a property of `fiat` with the name `color` and the value "Medium Blue".

How to change a property. You can change the value of a property at any time. All you need to do is assign the property to a new value. Like, let's say you wanted to set the mileage of your nifty Fiat to an even 10,000. You'd do that like this:

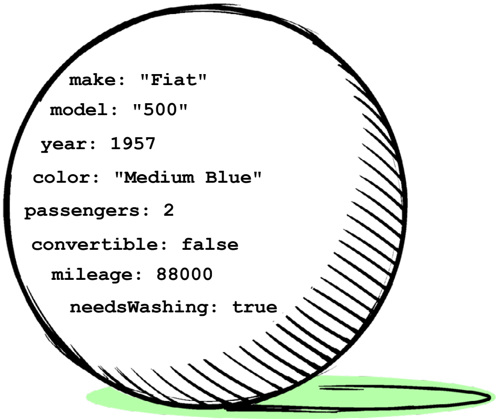
```
fiat.mileage = 10000;
```

Just specify the property you want to change and then give it a new value. Note: in some states this may be illegal!

You can extend your object at any time with new properties. To do this, you just specify the new property and give it a value. For instance, say you wanted to add a boolean that indicates when the Fiat needs to be washed:

```
fiat.needsWashing = true;
```

As long as the property doesn't already exist in the object, it's added to the object. Otherwise, the property with this name is updated.

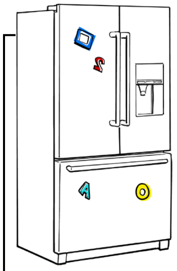


The new property is added to your object.

How to compute with properties. Computing with properties is simple: just use a property like you would any variable (or any value). Here are a few examples:

You can use an object's property just like you use a variable, except you need to use dot notation to access the property in the object.

```
if (fiat.year < 1965) {
    classic = true;
}
for (let i = 0; i < fiat.passengers; i++) {
    addPersonToCar();
}
```



Object Magnets

This code got all scrambled up on the fridge. Practice your object creating and dot notation skills by putting it all back together. Be careful, some extra magnets might have gotten mixed in!

`age`

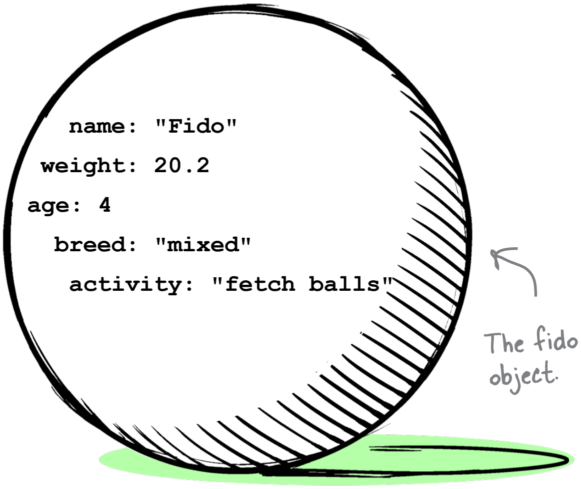
`bark` `,` `20.2` `fido.activity` `weight` `"Fido"`

`fido.weight` `Fido` `fido.bark` `"fetch balls"` `breed`

`,` `,` `,` `4` `fido.name`

Use these magnets to complete the code.

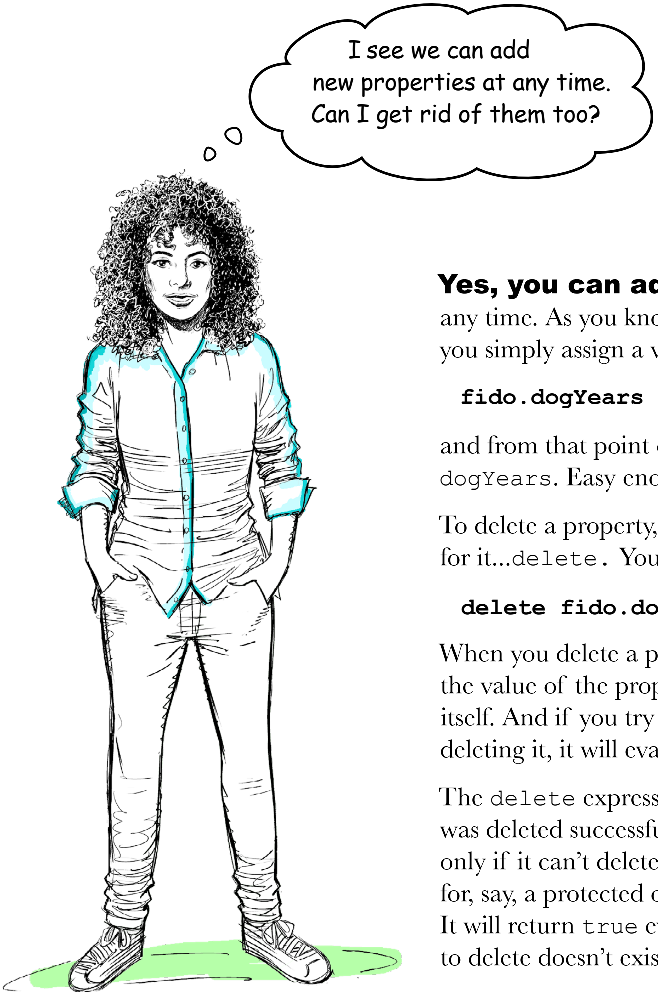
```
let fido = {
    name: _________
    _________: 20.2
    age: ________
    ________: "mixed",
    activity: ____________
};
let bark;
if (_____________ > 20) {
   bark = "WOOF WOOF";
} else {
   bark = "woof woof";
}
let speak = __________ + " says " + ________ + " when he wants to " + ____________;
console.log(speak);
```



name: "Fido"
weight: 20.2
age: 4
breed: "mixed"
activity: "fetch balls"

The fido object.

Solution on page 230


I see we can add new properties at any time. Can I get rid of them too?

Yes, you can add or delete properties at any time. As you know, to add a property to an object you simply assign a value to a new property, like this:

```
fido.dogYears = 35;
```

and from that point on `fido` will have a new property, `dogYears`. Easy enough.

To delete a property, we use a special keyword...wait for it...`delete`. You use the `delete` keyword like this:

```
delete fido.dogYears;
```

When you delete a property, you're not just deleting the value of the property, you're deleting the property itself. And if you try to use `fido.dogYears` after deleting it, it will evaluate to `undefined`.

The `delete` expression returns `true` if the property was deleted successfully. `delete` will return `false` only if it can't delete a property (which could happen for, say, a protected object that belongs to the browser). It will return `true` even if the property you're trying to delete doesn't exist in the object.

there are no Dumb Questions

Q: How many properties can an object have?

A: As few or as many as you want. You can have an object with no properties, or you can have an object with hundreds of properties. It's really up to you.

Q: How can I create an object with no properties?

A: Just like you create any object, only leave out all the properties. Like this:

```
let lookMaNoProps = { };
```

Q: I know I just asked how to create an object with no properties, but why would I want to do this?

A: Well, you might want to start with an entirely empty object and then add your own properties dynamically, depending on the logic of your code. Here's an example:

```
let lookMaNoProps = { };
lookMaNoProps.age = 10;
if (lookMaNoProps.age > 5) {
  lookMaNoProps.school = "Elementary";
}
```

Q: What's better about an object than just using a bunch of variables? After all, each of the properties in the fiat object could just be its own variable, right?

A: Objects package up your data so that you can focus on the high-level design of your code, not the nitty-gritty details. Say you want to write a traffic simulator with dozens of cars; you'll want to focus on cars and streetlights and road objects and not hundreds of little variables. Objects also make your life easier because they encapsulate, or hide, the complexity of the state and behavior of your objects so you don't have to worry about them. How all that works will become much clearer as you gain experience with objects.

Q: If I try to add a new property to my object, and the object already has a property with that name, what happens?

A: If you try to add a new property, like needsWashing, to fiat and fiat already has a property needsWashing, then you'll be changing the existing value of the property. So if you say:

```
fiat.needsWashing = true;
```

but fiat already contains a property needsWashing with the value false, then you're changing the value to true.

Q: What happens if I try to access a property that doesn't exist? Like if I said:

```
if (fiat.make) { ... }
```

but fiat didn't have a property make?

A: The result of the expression fiat.make will be undefined if fiat doesn't have a property named make.

Q: What happens if I put a comma after the last property?

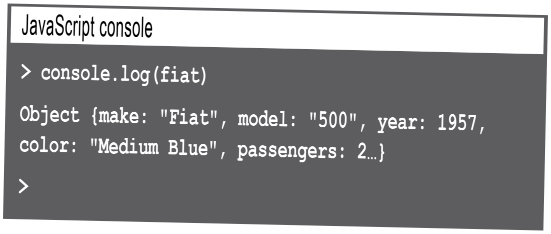
A: In modern browsers it won't cause an error. However, in older versions of some browsers this will cause your JavaScript to halt execution.

Q: Can I use console.log to display an object in the console?

A: You can. Just write:

```
console.log(fiat);
```

in your code, and when you load the page with the console open, you'll see information about the object displayed in the console.



JavaScript console

```
> console.log(fiat)
Object {make: "Fiat", model: "500", year: 1957,
color: "Medium Blue", passengers: 2…}
>
```

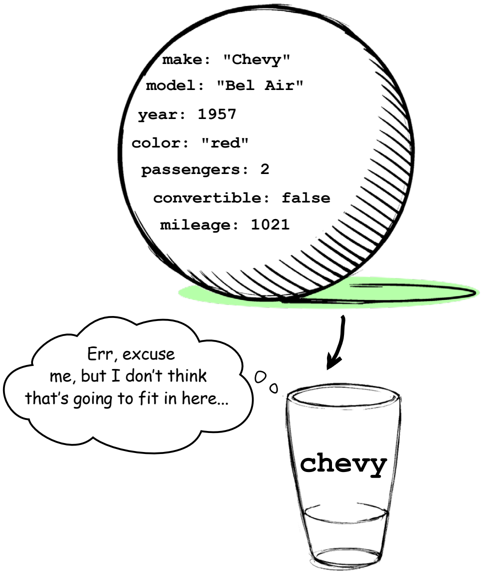
You can also use console.table(fiat)...try it!

How does a variable hold an object? Inquiring minds want to know...

PRODUCTION: Head First
SCENE 2
TAKE 3
ROLL
DATE
SOUND
PRODUCTION CO. O'Reilly
DIRECTOR
Behind the Scenes

You've already seen that a variable is like a container, and it holds a value. But numbers, strings, and booleans are pretty small values. What about objects? Can a variable hold any sized object, no matter how many properties you put in it?

- **Variables don't actually hold objects.**
- **Instead, they hold a reference to an object.**
- **The reference is like a pointer to the actual object.**
- **In other words, a variable doesn't hold the object itself, but it holds something like a pointer to the object. And, in JavaScript, we don't really know *what* is inside a reference variable. We *do* know that whatever it is, it points to our object.**
- **When we use dot notation, the JavaScript interpreter takes care of using the reference to get the object and then accesses its properties for us.**


make: "Chevy"
model: "Bel Air"
year: 1957
color: "red"
passengers: 2
convertible: false
mileage: 1021
Err, excuse me, but I don't think that's going to fit in here...
chevy

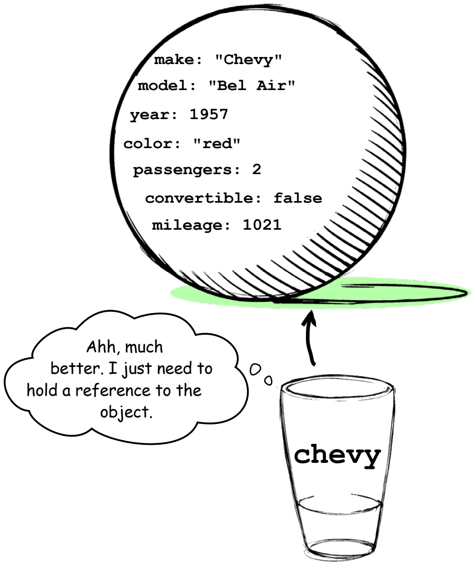
So, you can't stuff an object into a variable. We often think of it that way, but that's not what happens; variables aren't giant expandable cups that can grow to the size of any object. Instead, an object variable just holds a reference to the object.

Here's another way to look at it: a primitive variable represents the actual *value* of the variable, while an object variable represents *a way to get to the object*. In practice you'll only need to think of objects as, well, objects, like dogs and cars, not as references, but knowing variables contain *references* to objects will come in handy later (and we'll see that in just a few pages).

And also think about this: you use dot notation (.) on a reference variable to say "use the reference *before* the dot to get me the object that has the property *after* the dot." (Read that sentence a few times and let it sink in.) For example:

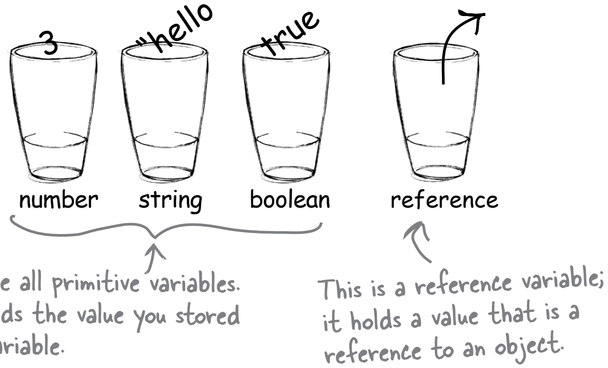
```
car.color;
```

means "use the object referenced by the variable `car` to access the `color` property."


make: "Chevy"
model: "Bel Air"
year: 1957
color: "red"
passengers: 2
convertible: false
mileage: 1021
Ahh, much better. I just need to hold a reference to the object.
chevy

Comparing primitives and objects

Think of an object reference as just another variable value, which means that we can put that reference in a cup, just like we can primitive values. With primitive values, the value of a variable is… the *value*, like 5, –26.7, "hi", or false. With reference variables, the value of the variable is a *reference*: a value that represents a way to get to a specific object.



These are all primitive variables. Each holds the value you stored in the variable.

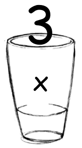
This is a reference variable; it holds a value that is a reference to an object.

Initializing a primitive variable

When you declare and initialize a primitive, you give it a value, and that value goes right in the cup, like this:

```
let x = 3;
```

The variable holds the number three.



A number value.

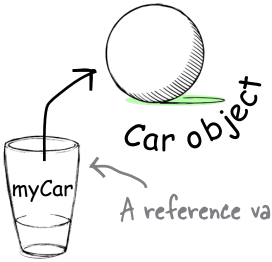
Initializing an object (reference) variable

When you declare and initialize an object, you make the object using object notation, but that object won't fit in the cup. So what goes in the cup is a *reference* to the object:

```
let myCar = {...};
```

A reference to the car object goes into the variable.

The car object itself does not go into the variable!



A reference value.

Behind the Scenes

We don't know (or care) how the JavaScript interpreter represents object references.

We just know we can access an object and its properties using dot notation.

Doing even more with objects

Let's say you're looking for a good car for your stay in Webville. Your criteria? How about:

☐ Built in 1960 or before

☐ 10,000 miles or less

You also want to put your new coding skills to work (and make your life easier), so you want to write a function that will "prequalify" cars for you—that is, if the car meets your criteria, then the function returns true; otherwise, the car isn't worth your time and the function returns false.

More specifically, you're going to write a *function* that *takes a car object as an argument* and puts that car through the test, returning a boolean value. Your function is going to work for *any car object*.

Let's give it a shot:

Here's the function.

You're going to pass it a car object.

```
function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}
```

Just use dot notation on the car parameter to access the mileage and year properties.

Test each property value against the prequalification criteria.

If either of the disqualification tests succeeds, we return false. Otherwise, we return true, meaning we've successfully prequalified!

Now let's give this function a try. First, you need a car object. How about this one:

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

What do you think? Should we consider this taxi? Why or why not?

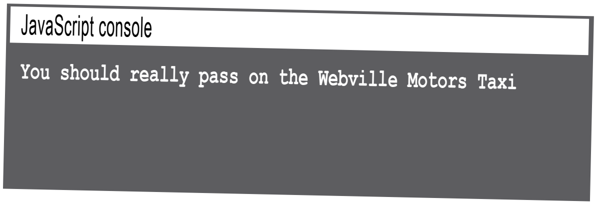
Doing some prequalification

We've done enough talking about objects. Let's actually create one and put it through its paces using the `prequal` function. Grab your favorite basic HTML page ("prequal.html") and throw in the code below. Then load the page and see if the taxi qualifies:

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};

function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}

let worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

Does the taxi cut it?

Here's what we got. Let's quickly trace through the code on the next page to see how the taxi got rejected...

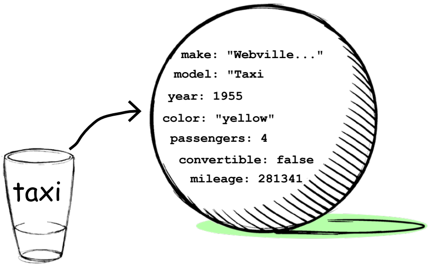


```
You should really pass on the Webville Motors Taxi
```

Stepping through prequalification

① First, we create the taxi object and assign it to the variable `taxi`. Of course, the `taxi` variable holds a reference to the taxi object, not the object itself.

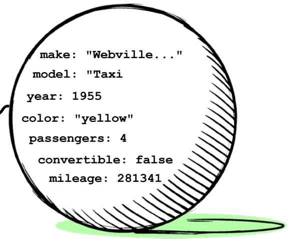
```
let taxi = { ... };
```


taxi
make: "Webville..."
model: "Taxi
year: 1955
color: "yellow"
passengers: 4
convertible: false
mileage: 281341

② Next, we call `prequal`, passing it the value in the argument `taxi`, which is assigned to the parameter `car`.

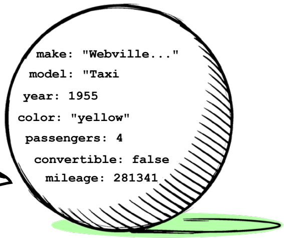
```
function prequal(car) {
    ...
}
```

car points to the same object as taxi!


make: "Webville..."
model: "Taxi
year: 1955
color: "yellow"
passengers: 4
convertible: false
mileage: 281341

③ We then perform the tests in the body of the function, using the taxi object in the `car` parameter.

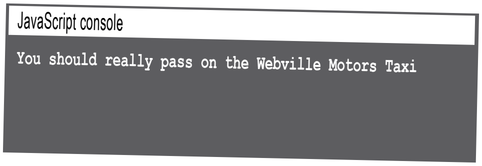
```
if (car.mileage > 10000) {
    return false;
} else if (car.year > 1960) {
    return false;
}
```


make: "Webville..."
model: "Taxi
year: 1955
color: "yellow"
passengers: 4
convertible: false
mileage: 281341

In this case, the taxi's mileage is way above 10,000 miles, so prequal returns false. Too bad; it's a cool ride.

④ Unfortunately, the taxi has a lot of miles, so the first test of car.mileage > 10000 is true. The function returns false, so `worthALook` is set to false. We then get "You should really pass on the Webville Motors Taxi" displayed in the console.

The prequal function returns false, and so we get...

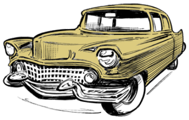

JavaScript console

```
You should really pass on the Webville Motors Taxi
```

```
let worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

Sharpen your pencil

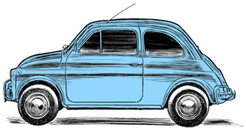
Your turn. Here are three more car objects; what is the result of passing each car to the prequal function? Work out the answers by hand, and then write the code to check your answers.



```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};

prequal(cadi);
```

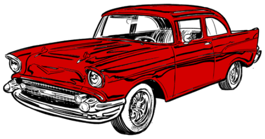
____

Write the value of prequal here.



```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

____



```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};

prequal(chevy);
```

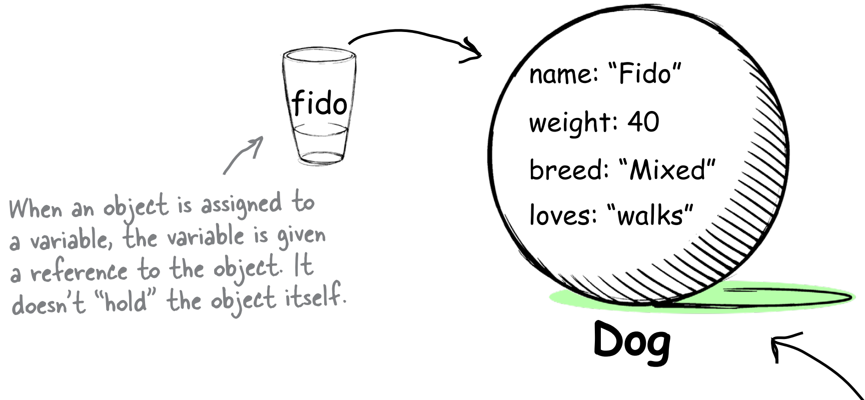
____

→ Solution on page 231

Let's talk a little more about passing objects to functions

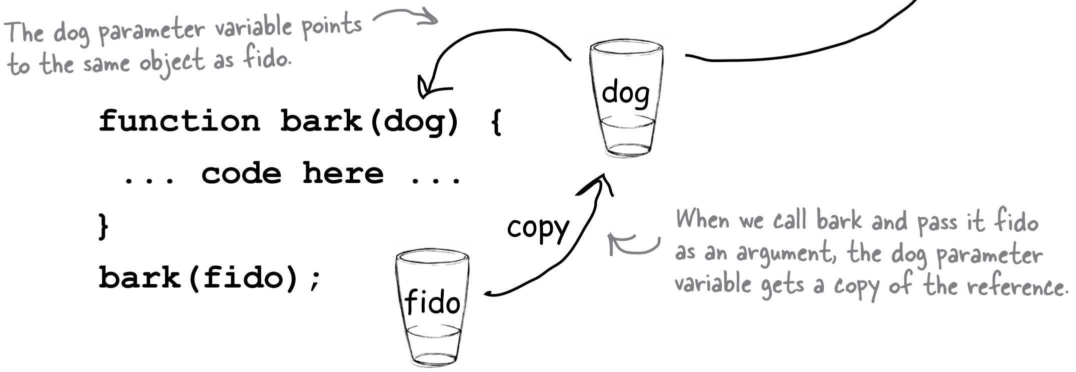
We've already talked a bit about how arguments are passed to functions—arguments are *passed by value*, which means *pass-by-copy*. So if we pass an integer, the corresponding function parameter gets a copy of the value of that integer for its use in the function. The same rules hold true for objects; however, we should look a little more closely at what pass-by-value means for objects to understand what happens when you pass an object to a function.

You already know that when an object is assigned to a variable, that variable holds a *reference* to the object, not the object itself. Again, think of a reference as a pointer to the object:



When an object is assigned to a variable, the variable is given a reference to the object. It doesn't "hold" the object itself.

So, when you call a function and pass it an object, you're passing the *object reference*, not the object itself. Using our pass-by-value semantics, a copy of the reference is passed into the parameter, and that reference remains a pointer to the original object.



The dog parameter variable points to the same object as fido.

```
function bark(dog) {
 ... code here ...
}
bark(fido);
```

When we call bark and pass it fido as an argument, the dog parameter variable gets a copy of the reference.

So what does this all mean? Well, one of the biggest ramifications is that if you change a property of the object in a function, you're changing the property in the *original* object. So, any changes you make to the object inside a function will still be there when the function completes. Let's step through an example...

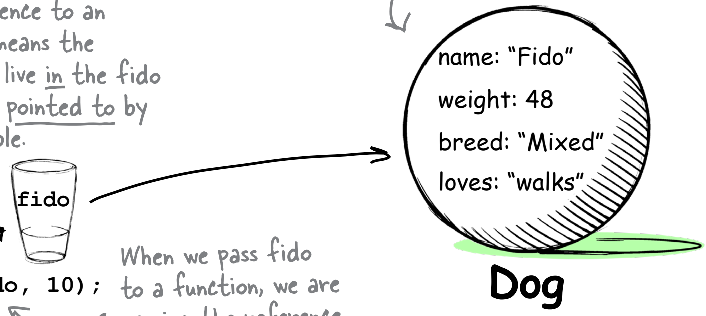
Putting Fido on a diet

Let's say we are testing a new method of weight loss for dogs, which we want to neatly implement in a function named `loseWeight`. All you need to do is pass `loseWeight` your dog object and an amount to lose, and like magic, the dog's weight will be reduced. Here's how it works:

① First, check out the dog object, fido, which we are going to pass to the loseWeight function:

fido is a reference to an object, which means the object doesn't live in the fido variable, but is pointed to by the fido variable.

Here's the dog object.



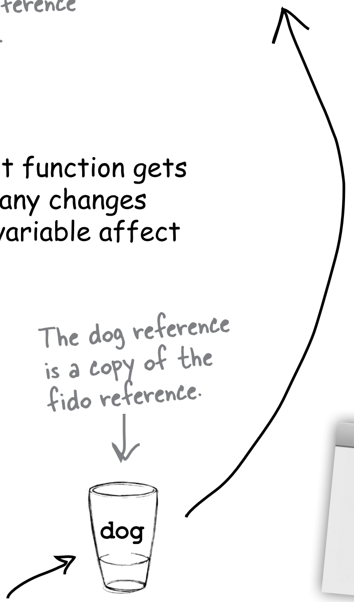
```
loseWeight(fido, 10);
```

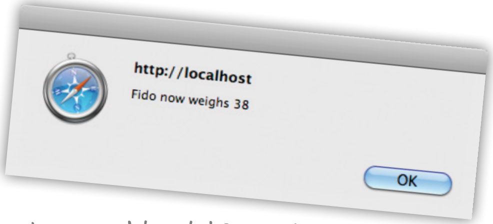
When we pass fido to a function, we are passing the reference to the object.

② The dog parameter of the loseWeight function gets a copy of the reference to fido. So, any changes to the properties of the parameter variable affect the object that was passed in.

When we pass fido into loseWeight, what gets assigned to the dog parameter is a copy of the reference, not a copy of the object. So fido and dog point to the same object.

The dog reference is a copy of the fido reference.





Fido now weighs 38

```
function loseWeight(dog, amount) {
  dog.weight = dog.weight - amount;
}

alert(fido.name + " now weighs " + fido.weight);
```

So, when we subtract 10 pounds from dog.weight, we're changing the value of fido.weight.

Sharpen your pencil

You've been given a super-secret file and two functions that allow access to get and set the contents of the file, but only if you have the right password. The first function, getSecret, returns the contents of the file if the password is correct and logs each attempt to access the file. The second function, setSecret, updates the contents of the file and resets the access tracking to 0. It's your job to fill in the blanks below to complete the JavaScript and test your functions.

```
function getSecret(file, secretPassword) {
    _______.opened = _______.opened + 1;
    if (secretPassword == _______.password) {
        return ______.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == _______.password) {
        ______.opened = 0;
        ______.contents = secret;
    }
}

let superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
let secret = getSecret(_______________, _____);
console.log(secret);

setSecret(_________________, _____, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(_______________, _____);
console.log(secret);
```

→ Solution on page 232

The Auto-O-Matic

TheAuto-O-Matic is a variation of the Phrase-O-Matic from Chapter 4, this time designed to sell cars.

```
<!doctype html>
<html lang="en">
<head>
  <title>Auto-O-Matic</title>
  <meta charset="utf-8">
  <script>
    function makeCar() {
        let makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
        let models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
        let years = [1955, 1957, 1948, 1954, 1961];
        let colors = ["red", "blue", "tan", "yellow", "white"];
        let convertible = [true, false];

        let rand1 = Math.floor(Math.random() * makes.length);
        let rand2 = Math.floor(Math.random() * models.length);
        let rand3 = Math.floor(Math.random() * years.length);
        let rand4 = Math.floor(Math.random() * colors.length);
        let rand5 = Math.floor(Math.random() * 5) + 1;
        let rand6 = Math.floor(Math.random() * convertible.length);

        let car = {
            make: makes[rand1],
            model: models[rand2],
            year: years[rand3],
            color: colors[rand4],
            passengers: rand5,
            convertible: convertible[rand6],
            mileage: 0
        };
        return car;
    }

    function displayCar(car) {
        console.log("Your new car is a " + car.year + " " + car.make + " " + car.model);
    }

    let carToSell = makeCar();
    displayCar(carToSell);

  </script>
</head>
<body></body>
</html>
```

The Auto-O-Matic is similar to the Phrase-O-Matic from Chapter 4, except that the words are car properties, and we're generating a new car object instead of a marketing phrase!

Check out what it does and how it works.

The Auto-O-Matic

Brought to you by the same guy who brought you the Phrase-O-Matic, the Auto-O-Matic creates knock-off cars all day long. That is, instead of generating marketing messages, this code generates makes, models, years, and all the other properties of a car object. *It's your very own car factory in code.* Let's take a closer look at how it works:

① First, we have a `makeCar` function that we can call whenever we want to make a new car. We've got four arrays with the makes, models, years, and colors of cars, and an array with true and false options for whether a car is a convertible. We generate five random numbers so we can pick a make, a model, a year, a color, and whether the car is a convertible randomly from these five arrays. And we generate one more random number we're using for the number of passengers:

We have several makes, models, years, and colors to choose from in these four arrays...

```
let makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
let models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
let years = [1955, 1957, 1948, 1954, 1961];
let colors = ["red", "blue", "tan", "yellow", "white"];
let convertible = [true, false];
```

...and we'll use this array to choose a convertible property value, either true or false.

We're going to combine values from the arrays randomly using these four random numbers.

```
let rand1 = Math.floor(Math.random() * makes.length);
let rand2 = Math.floor(Math.random() * models.length);
let rand3 = Math.floor(Math.random() * years.length);
let rand4 = Math.floor(Math.random() * colors.length);
let rand5 = Math.floor(Math.random() * 5) + 1;
let rand6 = Math.floor(Math.random() * convertible.length);
```

We'll use this random number for the number of passengers. We're adding 1 to the random number so we can have at least one passenger in the car.

And we'll use this random number to choose whether a car is convertible or not.

② Instead of creating a string by mixing and matching the various car properties, like we did with the Phrase-O-Matic, this time we're creating a new object, `car`. This car has all the properties you'd expect. We pick values for the `make`, `model`, `year`, and `color` properties from the arrays using the random numbers we created in step 1, and also add the `passengers`, `convertible`, and `mileage` properties.

```
let car = {
    make: makes[rand1],
    model: models[rand2],
    year: years[rand3],
    color: colors[rand4],
    passengers: rand5,
    convertible: convertible[rand6],
    mileage: 0
};
```

We're creating a new car object, with property values made from the values in the arrays.

We're also setting the number of passengers to the random number we created and setting the convertible property to true or false using the convertible array.

Finally, we're just setting the mileage property to 0 (it is a new car, after all).

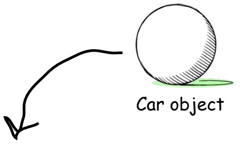
③ The last statement in `makeCar` returns the new `car` object:

```
return car;
```

Returning an object from a function is just like returning any other value. Let's now look at the code that calls `makeCar`:

```
function displayCar(car) {
  console.log("Your new car is a " + car.year + " " +
                car.make + " " + car.model);
}
let carToSell = makeCar();
displayCar(carToSell);
```

Don't forget; what you're returning (and assigning to the carToSell variable) is a reference to a car object.

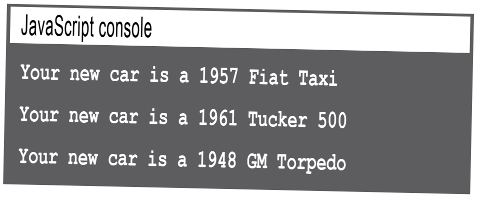


```
let carToSell = makeCar();
```

First, we call the `makeCar` function and assign the value it returns to `carToSell`. We then pass the `car` object returned from `makeCar` to the function `displayCar`, which simply displays a few of its properties in the console.

④ Go ahead and load up the Auto-O-Matic in your browser ("autoomatic.html") and give it a whirl. You'll find no shortage of new cars to generate, and remember there's a sucker born every minute.

Here's your new car! We think a '57 Fiat Taxi would be a cool car to have.



```
Your new car is a 1957 Fiat Taxi
Your new car is a 1961 Tucker 500
Your new car is a 1948 GM Torpedo
```

Reload the page a few times like we did!

Oh, behave! Or, how to add behavior to your objects

You didn't think objects were just for storing numbers and strings, did you? Objects are *active*. Objects can *do things*. Dogs don't just sit there...they bark, run, and play catch, and a dog object should too! Likewise, we drive cars, park them, put them in reverse, and make them brake. Given everything you've learned in this chapter, you're all set to add behavior to your objects. Here's how we do that:

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    drive: function() {
       alert("Zoom zoom!");
    }
};
```

You can add a function directly to an object like this.

All you do is assign a function definition to a property. Yup, properties can be functions too!

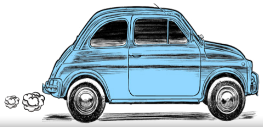
Notice we don't supply a name in the function definition; we just use the function keyword followed by the body. The name of the function is the name of the property.

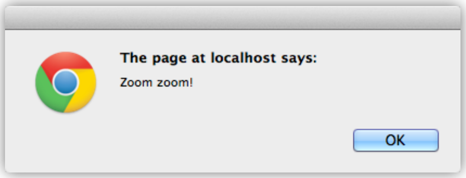
And a bit of nomenclature: we typically refer to functions inside an object as <u>methods</u>. That is a common object-oriented term for a function in an object.

To call the `drive` function—excuse us—to call the `drive` *method*, you use dot notation again, this time with the object name `fiat` and the property name `drive`, only you follow the property name with parentheses (just like you would when you call any other function).

```
fiat.drive();
```

We use dot notation to access the function in fiat, just like we would any other property. We say we're "calling the drive method in the fiat object."





The page at localhost says:

Zoom zoom!

The result of calling the fiat's drive method.

Improving the drive method

Let's make the `fiat` a little more car-like in behavior. Most cars can't be driven until the engine is started, right? How about we model that behavior? We'll need the following:

☐ **A boolean property to hold the state of the car (the engine is either on or off)**

☐ **A couple of methods to start and stop the car**

☐ **A conditional check in the drive method to make sure the car is started before we drive it**

We'll begin by adding a boolean `started` property along with methods to `start` and `stop` the car; then we'll update the `drive` method to use the `started` property:

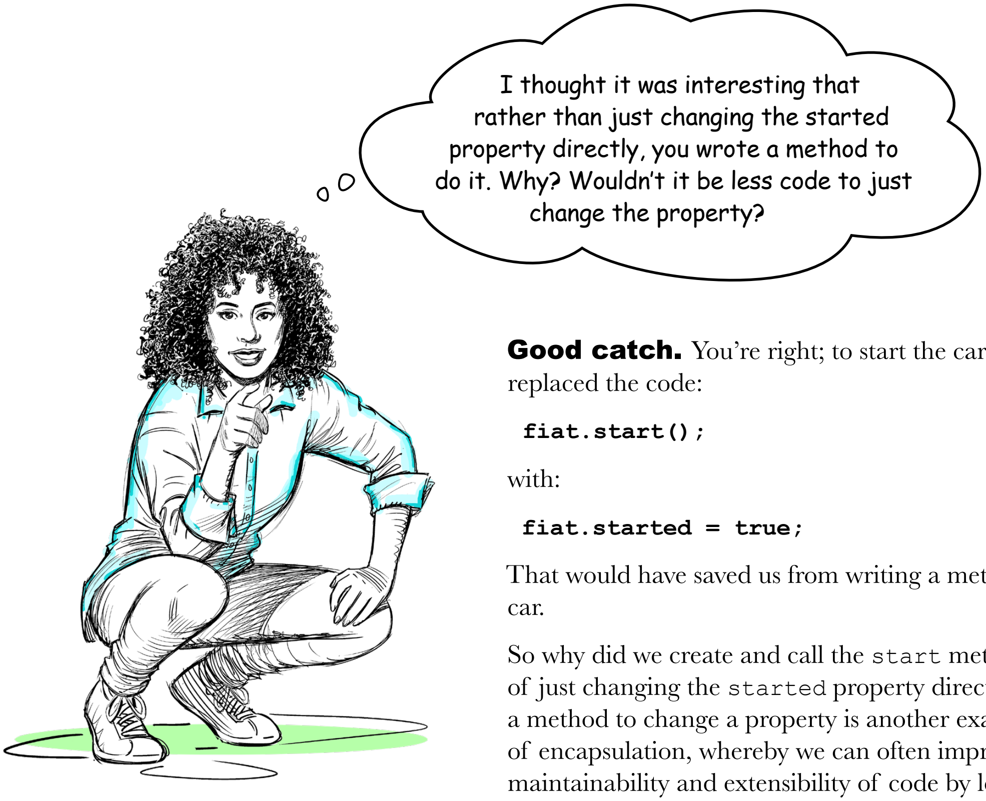
```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,

    start: function() {
       started = true;
    },

    stop: function() {
        started = false;
    },

    drive: function() {
        if (started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Here's the property to hold the current state of the engine (true if it is started and false if it is off).

Here's a method to start the car. All it does (for now) is set the started property to true.

And here's a method to stop the car. All it does is set the started property to false.

And here's where the interesting behavior happens: when we try to drive the car, if it is started we get a "Zoom zoom!" and if not, we get a warning that we should start the car first.


I thought it was interesting that rather than just changing the started property directly, you wrote a method to do it. Why? Wouldn't it be less code to just change the property?

Good catch. You're right; to start the car we could have replaced the code:

```
fiat.start();
```

with:

```
fiat.started = true;
```

That would have saved us from writing a method to start the car.

So why did we create and call the `start` method instead of just changing the `started` property directly? Using a method to change a property is another example of encapsulation, whereby we can often improve the maintainability and extensibility of code by letting an object worry about how it gets things done. It's better to have a `start` method that knows how to start the car than for you to have to know "to start the car we need to take the `started` property and set it to true."

Now, you may still be saying, "What's the big deal? Why not just set the property to true to start the car?!" Consider a more complex `start` method that checks the seatbelts, ensures there is enough fuel, checks the battery, checks the engine temperature, and so on, all before setting `started` to true. You certainly don't want to think about all that every time you start the car. You just want a handy method to call that gets the job done. By putting all those details into a method, we've created a simple way for you to get an object to do some work while letting the object worry about how it gets that work done.

Take the fiat for a test drive

Let's take our new and improved `fiat` object for a test drive. Let's give it a good testing—we'll try to drive it before it's started, and then start, drive, and stop it. To do that, make sure you have the code for the `fiat` object typed into a simple HTML page ("carWithDrive.html"), including the new methods `start`, `stop`, and `drive`, and then add this code below the object:

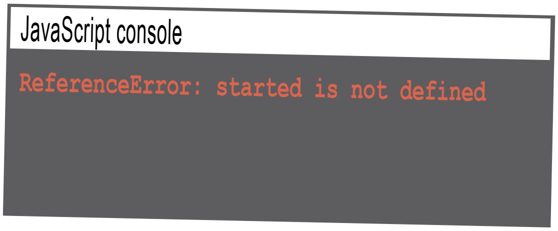
```
fiat.drive();
fiat.start();
fiat.drive();
fiat.stop();
```

First, we'll try to drive the car, which should give us a message to start the car. Then we'll start it for real, and we'll drive it. Finally, when we're done, we'll stop the car.

Go ahead and load the page in your browser and let the road trip begin!

Uh-oh, not so fast...

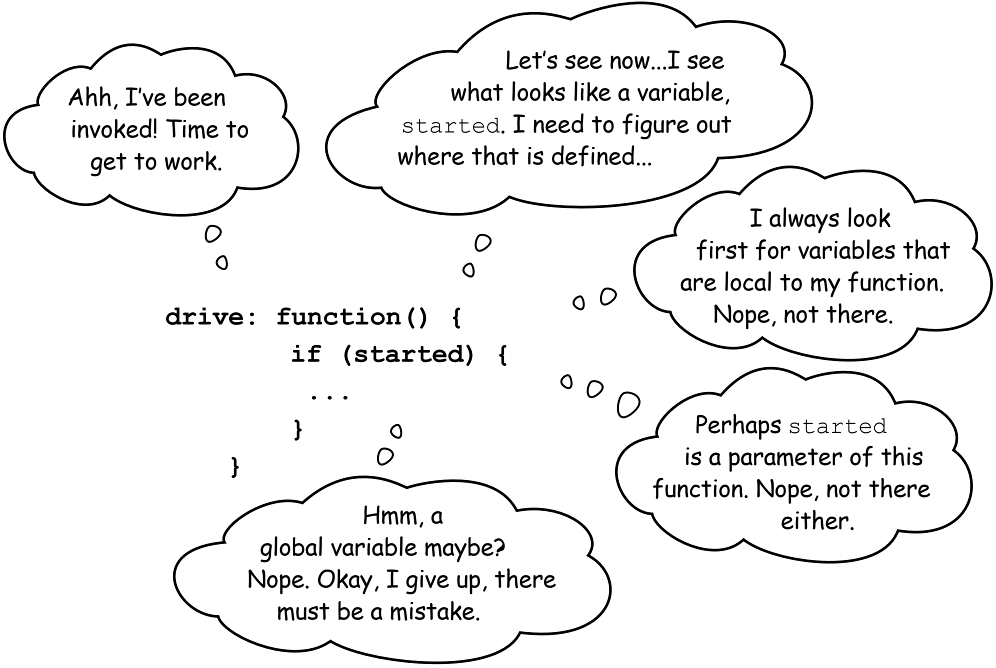
If you can't drive your Fiat, you're not alone. In fact, find your way to your JavaScript console and you're likely to see an error message similar to the one we got saying that `started` is not defined.



JavaScript console

```
ReferenceError: started is not defined
```

So, what's going on? Let's listen in on the `drive` method and see what's happening as we try to drive the car with `fiat.drive()`:



Ahh, I've been invoked! Time to get to work.
Let's see now...I see what looks like a variable, started. I need to figure out where that is defined...
I always look first for variables that are local to my function. Nope, not there.
Perhaps started is a parameter of this function. Nope, not there either.
Hmm, a global variable maybe? Nope. Okay, I give up, there must be a mistake.

```
drive: function() {
        if (started) {
          ...
        }
}
```

Why doesn't the drive method know about the started property?

Here's the conundrum: we've got references to the property `started` in the `fiat` object's methods, and normally when we're trying to resolve a variable in a function, that variable turns out to be a local variable, a parameter of the function, or a global variable. But in the `drive` method, `started` is none of those things; instead, it's a *property* of the `fiat` object.

Shouldn't this code just work, though? In other words, we wrote `started` in the `fiat` object; shouldn't JavaScript be smart enough to figure out we mean the `started` property?

Nope. As you can see, it isn't. How can that be?

Okay, here's the deal: what looks like a variable in the method is really a property of the object, but we aren't telling JavaScript which object. You might say to yourself, "Well, obviously we mean THIS object, this one right here! How could there be any confusion about that?" And, yes, we want the property of this very object. In fact, there's a keyword in JavaScript named `this`, and that is exactly how you tell JavaScript you mean *this object we're in*.

Really, if you want me to know which object `started` belongs to, you're going to have to tell me.

```
drive: function() {
        if (started) {
          ...
        }
    }
```

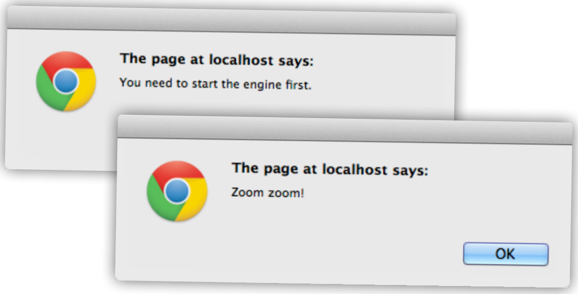
So, let's add the `this` keyword and get this code working:

```
let fiat = {
    make: "Fiat",
    // other properties are here, we're just saving space
    started: false,

    start: function() {
        this.started = true;
    },

    stop: function() {
        this.started = false;
    },

    drive: function() {
        if (this.started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Use this along with dot notation before each occurrence of the started property to tell the JavaScript interpreter you mean the property of THIS very object, rather than having JavaScript think you're referring to a variable.

A test drive with "this"

Go ahead and update your code, and take it for a spin! Here's what we got:



The page at localhost says:

You need to start the engine first.

The page at localhost says:

Zoom zoom!

BE the Browser

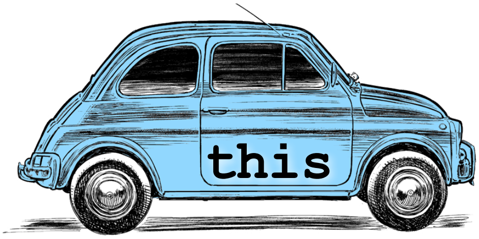
Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. After you've done the exercise, look at the end of the chapter to see if you found them all.

Go ahead and mark up the code right here...

```
let song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,

    play: function() {
        if (!playing) {
            this = true;
            console.log("Playing "
                + name + " by " + artist);
        }
    },

    pause: function() {
        if (playing) {
            this.playing = false;
        }
    }
};

this.play();
this.pause();
```

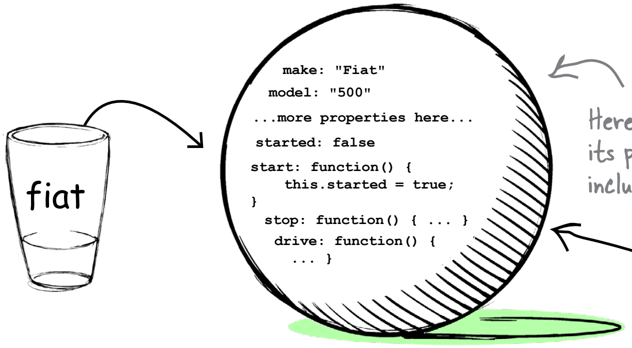
Solution on page 233

How "this" works



You can think of `this` like a variable that is assigned to the object whose method was just called. In other words, if you call the `fiat` object's `start` method with `fiat.start()` and use `this` in the body of the `start` method, then `this` will refer to the `fiat` object. Let's look more closely at what happens when we call the `start` method of the `fiat` object.

First, we have an object representing the Fiat car, which is assigned to the `fiat` variable:

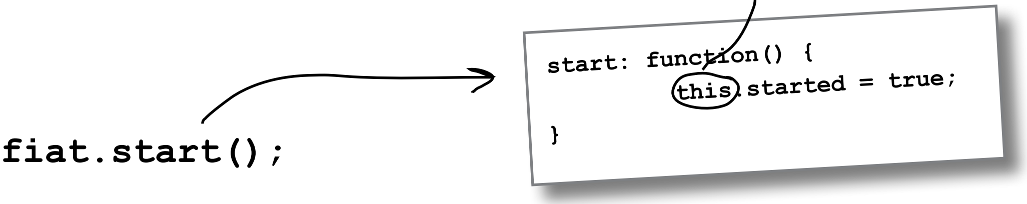


```
make: "Fiat"
model: "500"
...more properties here...
started: false
start: function() {
    this.started = true;
}
stop: function() { ... }
drive: function() {
   ... }
```

Here's the fiat object with all its property names and values, including the start method.

In this case, **this** refers to the fiat object, because we called the fiat object's start method.

Then, when we call the `start` method, JavaScript takes care of assigning `this` to the `fiat` object.



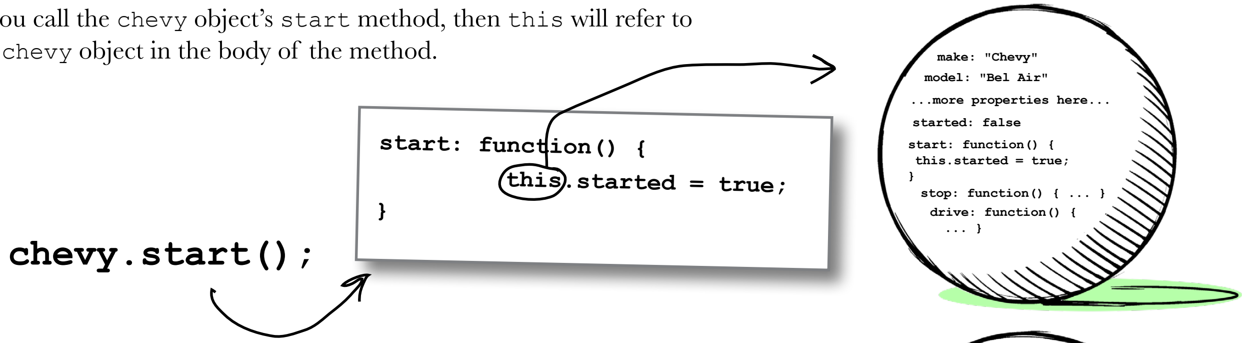
```
fiat.start();
```

```
start: function() {
            this.started = true;

}
```

Whenever we call a method in an object, **this** will refer to that object. So here, **this** refers to the fiat object.

The real key to understanding `this` is that whenever a method is called, in the body of that method you can count on `this` to be assigned to the *object whose method was called*. Just to drive the point home, let's try it on a few other objects…

If you call the `chevy` object's `start` method, then `this` will refer to the `chevy` object in the body of the method.

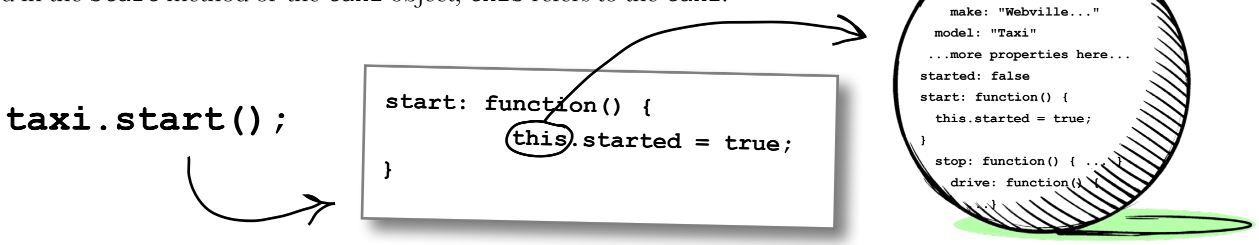


```
chevy.start();
```

```
start: function() {
          this.started = true;
}
```

```
make: "Chevy"
model: "Bel Air"
...more properties here...
started: false
start: function() {
 this.started = true;
}
 stop: function() { ... }
  drive: function() {
    ... }
```

And in the `start` method of the `taxi` object, `this` refers to the `taxi`.



```
taxi.start();
```

```
start: function() {
          this.started = true;
}
```

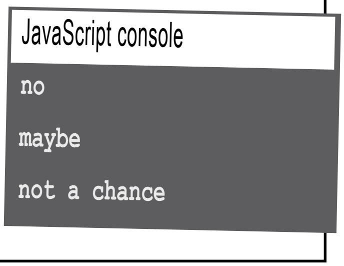
```
make: "Webville..."
model: "Taxi"
...more properties here...
started: false
start: function() {
  this.started = true;
}
 stop: function() { ... }
  drive: function() {
   ... }
```

Sharpen your pencil

Use your new **this** skills to help us finish this code. Check your answer at the end of the chapter.

```
let eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                          this.index = ________.index + 1;
                         if (_________.index  >= _________.advice.length) {
                            _________.index = 0;
                         }
                 },
                 look: function() {
                         return _________.advice[_________.index];
                 }
};
eightBall.shake();
console.log(eightBall.look());
```

Repeat this sequence several times to test your code.



```
no
maybe
not a chance
```

Solution on page 228

there are no Dumb Questions

Q: What's the difference between a method and a function?

A: A method is just a function that's been assigned to a property name in an object. You call functions using the function name, while you call methods using the object dot notation and the name of the property. You can also use the keyword **this** in a method to refer to the object whose method was called.

Q: I noticed that when using the "function" keyword within an object we don't give the function an explicit name. What happened to the function name?

A: Right. To call methods, we use the property name in the object rather than explicitly naming the function and using that name. For now, just take this as the convention we use, but later in the book we'll dive into the topic of anonymous functions (which is what we call functions that don't explicitly have names).

Q: Can methods have local variables, like functions can?

A: Yes. A method is a function. We just call it a method because it lives inside an object. So, a method can do anything a function can do precisely because a method is a function.

Q: So, you can return values from methods too?

A: Yes. What we said in the last answer!

Q: What about passing arguments to methods? Can we do that too?

A: Err, maybe you didn't read the answer two questions back? Yes!

Q: Can I add a method to an object after it's created, like I can with a property?

A: Yes. Think of a method as a function assigned to a property, so you can add a new one at any time:

```
// add a turbo method
car.engageTurbo =
        function() { ... };
```

Q: If I add a method like engageTurbo above, will the "this" keyword still work?

A: Yes. Remember, **this** is assigned the object whose method is called *at the time it is called*.

Q: When is the value of "this" set to the object? When we define the object, or when we call the method?

A: The value of **this** is set to the object when you call the method. So, when you call fiat.start(), **this** is set to fiat, and when you call chevy.start(), **this** is set to chevy. It *looks* like **this** is set when you define the object, because in fiat.start, **this** is always set to fiat, and in chevy.start, **this** is always set to chevy. But as you'll see later, there is a good reason the value of **this** is set when you call the method and not when you define the object. This is an important point we'll be coming back to a few different times.

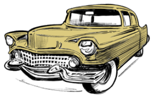
Brain Power

If you copy the start, stop, and drive methods into the chevy and cadi objects we created earlier, what do you have to change to make the methods work correctly?

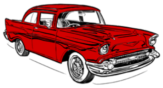
Answer: Nothing! this refers to "this object," the one whose method we're calling.

Exercise

It's time to get the whole fleet up and running. Add the drive method to each car object. When you've done that, add the code to start, drive, and stop each of them. Check your answers at the end of the chapter.



```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};
```



```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

Don't forget to add a comma after mileage when you add the new properties!

Add the started property and the methods to each car. Then use the code below to give them a test drive.

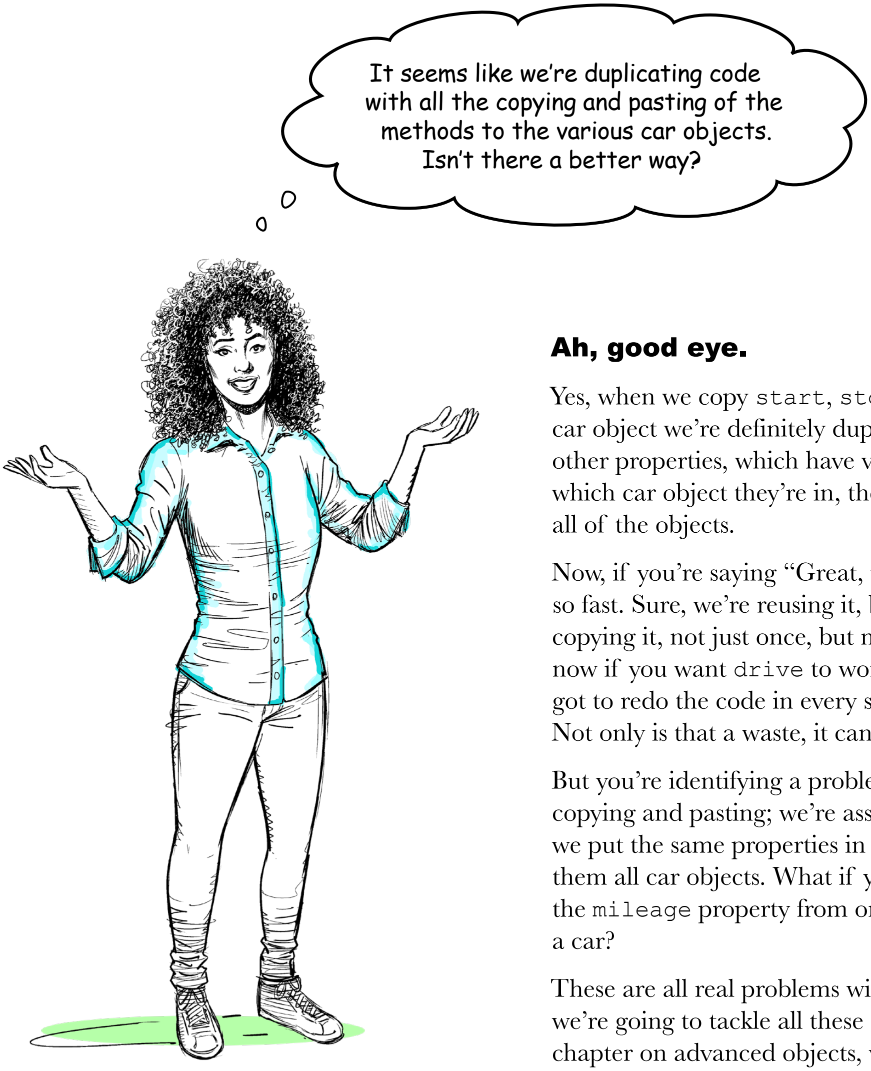
```
started: false,

start: function() {
   this.started = true;
},

stop: function() {
   this.started = false;
},

drive: function() {
   if (this.started) {
      alert(this.make + " " +
            this.model + " goes zoom zoom!");
   } else {
      alert("You need to start the engine first.");
   }
}
```

We improved the drive method just a bit, so make sure you get this new code.

Throw this code after the car object definitions to give them all a test drive.

```
cadi.start();
cadi.drive();
cadi.stop();
chevy.start();
chevy.drive();
chevy.stop();
taxi.start();
taxi.drive();
taxi.stop();
```

Solution on page 234


It seems like we're duplicating code with all the copying and pasting of the methods to the various car objects. Isn't there a better way?

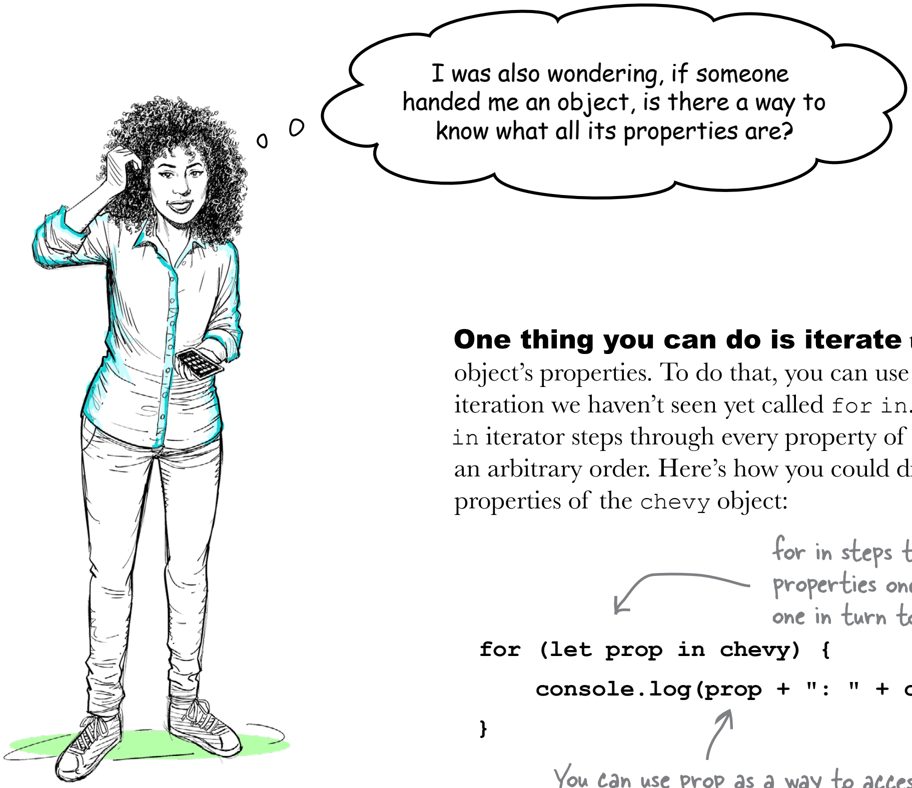
Ah, good eye.

Yes, when we copy `start`, `stop`, and `drive` into each car object we're definitely duplicating code. Unlike the other properties, which have values that depend on which car object they're in, the methods are the same for all of the objects.

Now, if you're saying "Great, we're reusing code!"...not so fast. Sure, we're reusing it, but we're doing that by copying it, not just once, but many times! What happens now if you want `drive` to work differently? Then you've got to redo the code in every single car object. Not good. Not only is that a waste, it can be error-prone.

But you're identifying a problem even larger than simple copying and pasting; we're assuming that just because we put the same properties in all our objects, that makes them all car objects. What if you accidentally leave out the `mileage` property from one of the objects—is it still a car?

These are all real problems with our code so far, and we're going to tackle all these questions in an upcoming chapter on advanced objects, where we'll talk about some techniques for properly reusing the code in your objects.

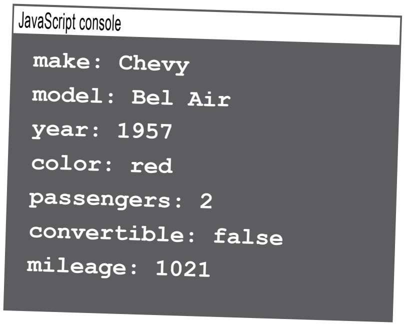

I was also wondering, if someone handed me an object, is there a way to know what all its properties are?

One thing you can do is iterate through an object's properties. To do that, you can use a form of iteration we haven't seen yet called `for in`. The `for in` iterator steps through every property of an object in an arbitrary order. Here's how you could display all the properties of the `chevy` object:

for in steps through the object's properties one at a time, assigning each one in turn to the variable prop.

```
for (let prop in chevy) {
    console.log(prop + ": " + chevy[prop]);
}
```

You can use prop as a way to access the property using bracket notation.



```
make: Chevy
model: Bel Air
year: 1957
color: red
passengers: 2
convertible: false
mileage: 1021
```

This brings up another topic: there's another way to access properties. Did you catch the alternative syntax we just used to access the properties of the `chevy` object? As it turns out, you've got two options when accessing a property of an object. You already know dot notation:

```
chevy.color
```

We just use the object name followed by a dot and a property name.

But there's another way, **bracket notation**, which looks like this:

```
chevy["color"]
```

Here we use the object name followed by brackets that enclose a property name in quotes.

Looks a bit like how we access array items.

The thing to know about these two forms is that they are equivalent and do the same thing. The only difference to be aware of is that bracket notation sometimes allows a little more flexibility, because you can make the property name an expression, like this:

```
chevy["co" + "lor"]
```

As long as the expression evaluates to a property name represented by a string, you can put any expression you want inside the brackets.

Method shorthand

While we're on the topic of syntax...there's a shorthand for declaring methods that we haven't told you about yet. You can drop the keyword `function`, and instead of writing your method declarations like this:

```
let fiat = {
    ...
    stop: function() {
        this.started = false;
    },
    ...
};
```

Note here we have that verbose function keyword, even though it is obviously a function.

I'm loving the simplified syntax!

you can write them like this:

```
let fiat = {
    ...
    stop() {
        this.started = false;
    },
    ...
};
```

And here it is nicely cleaned up. It still looks like a function, but we don't need the function keyword anymore.

Notice we also dropped the colon and moved the parentheses against the function name. That really cleans things up.

Now, you'll still call the method in the same way:

```
fiat.stop();
```

and this new syntax doesn't affect the method's semantics in any way.

Exercise

Go ahead and update your code for the cadi, chevy, and taxi objects to use the method shorthand. Make sure your code still runs, and check your solution at the end of the chapter before you go on. We'll use the shorthand for methods for the rest of the chapter.

→ Solution on page 235

How behavior affects state

Adding some gas-o-line

Objects have *state* and *behavior*. An object's properties allow us to keep state about the object—like its fuel level, its current temperature, or, say, the current song that is playing on the radio. An object's methods allow it to have behavior—like starting a car, turning up the heat, or fast-forwarding the playback of a song. Have you also noticed these two *interact*? Like, we can't start a car if it doesn't have fuel, and the amount of fuel should get reduced as we drive the car. Kinda like real life, right?

Let's play with this concept a little more by giving our cars some fuel, and then we can start to add interesting behavior. To add fuel, we'll add a new property, `fuel`, and a new method, `addFuel`. The `addFuel` method will have a parameter, `amount`, which we'll use to increase the amount of fuel in the `fuel` property. Let's add these properties to the `fiat` object:

```
let fiat = {
    make: "Fiat",
    model: "500",
    // other properties go here, we're saving some paper...
    started: false,
    fuel: 0,

    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
      if (this.started) {
        alert(this.make + " " + this.model + " goes zoom zoom!");
      } else {
        alert("You need to start the engine first.");
      }
    },
    addFuel(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

We've added a new property, fuel, to hold the amount of fuel in the car. The car will begin life on empty.

We're now using our new syntactic shorthand on our functions. Notice how this now looks quite different in contrast to the property definitions.

Let's also add a method, addFuel, to add fuel to the car. We can add as much fuel as we like by specifying the amount when we call the method.

Note: the shorthand way to write a method works fine with methods that have parameters too.

Remember, fuel is an object property, so we need the **this** keyword...

...but amount is a function parameter, so we don't need **this** to use it.

Now let's affect the behavior with the state

Now that we have fuel, we can start to implement some interesting behaviors. For instance, if there's no fuel, we shouldn't be able to drive the car! So, let's start by tweaking the `drive` method a bit to check the fuel level to make sure we've got some, and then we'll subtract one from `fuel` each time the car is driven. Here's the code to do that:

```
let fiat = {
    // other properties and methods here...
    drive() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
               this.fuel = this.fuel - 1;
            } else {
                alert("Uh-oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

Now we can check to make sure there's fuel before we drive the car. And, if we can drive the car, we should reduce the amount of fuel left each time we drive.

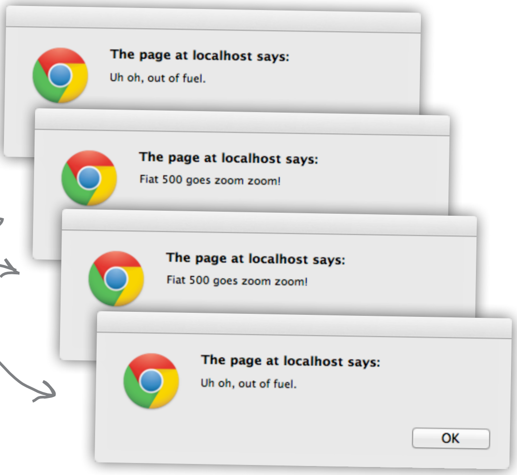
If there's no fuel left, we display a message and stop the engine. To drive the car again, you'll have to add fuel and restart the car.

Gas up for a test drive

Go ahead and update your code, and take it for a spin!
Here's what we got with the following test code:

```
fiat.start();
fiat.drive();
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.drive();
fiat.drive();
fiat.stop();
```

First, we tried to drive the car with no fuel; then we added some fuel and drove it until we ran out of fuel again! Try adding your own test code and make sure it works like you think it should.



The page at localhost says:
Uh oh, out of fuel.

The page at localhost says:
Fiat 500 goes zoom zoom!

The page at localhost says:
Fiat 500 goes zoom zoom!

The page at localhost says:
Uh oh, out of fuel.

Exercise

We still have some more work to do to fully integrate the fuel property into the car. For instance, should you be able to start the car if there's no fuel? Check out the start method:

```
start() {
    this.started = true;
}
```

It certainly looks like we can.

Help us integrate the fuel property into this code by checking the fuel level before the car is started. If there's no fuel and the start method is called, let the driver know with a handy alert like **`"The car is on empty, fill up before starting!"`** Rewrite the start method below, and then add it to your code and test it. Check your answer at the end of the chapter before you go on.

Your code here.

Solution on page 236

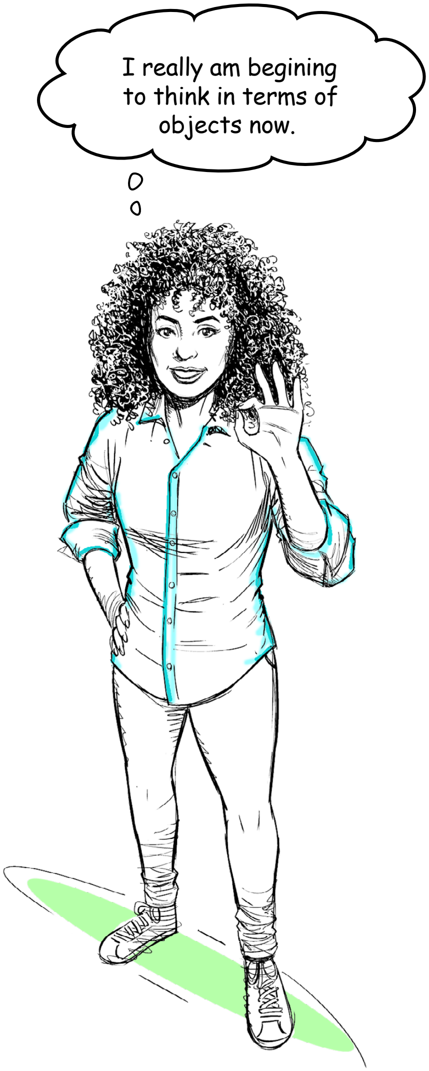
Brain Power

Take a look at all the fiat code. Are there other places you could use the fuel property to alter the car's behavior (or create behavior to modify the fuel property)? Jot down your ideas below.

Congrats on your first objects!

You've made it through the first objects chapter, and you're ready to move forward. Remember how you began with JavaScript? You were thinking of the world in terms of low-level numbers and strings and statements and conditionals and for loops and so on. Look how far you've come. You're starting to think at a higher level, and in terms of objects and methods. Just look at this code:

```
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.stop();
```


I really am begining to think in terms of objects now.

It's so much easier to understand what's going on in this code, because it describes the world as a set of objects with state and behavior.

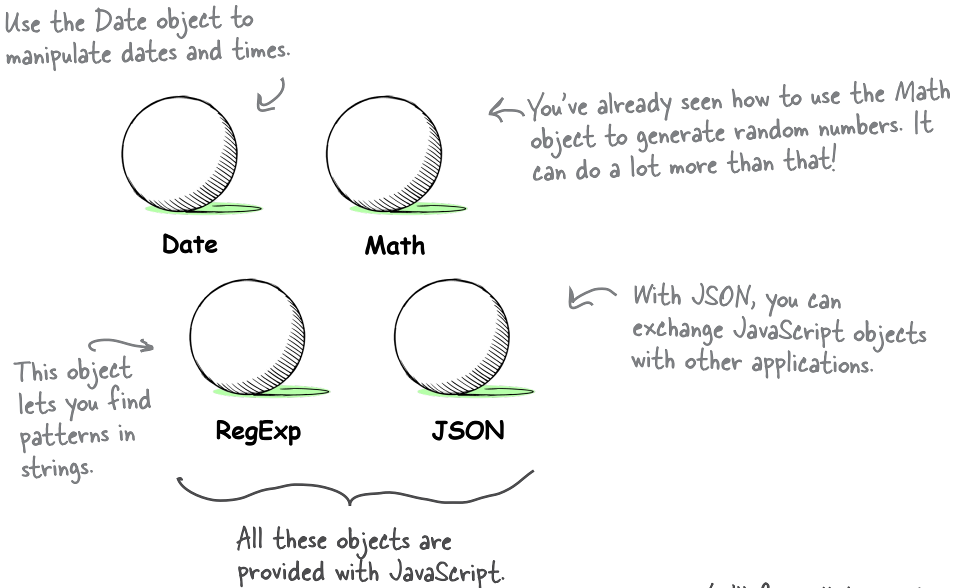
And this is just the beginning. You can take it so much further, and we will. Now that you know about objects, we're going to keep developing your skills to write truly object-oriented code using even more features of JavaScript and quite a few best practices (which become oh-so-important with objects).

But there's one more thing you should know, before you leave this chapter…

Guess what? There are objects all around you!

(And they'll make your life easier)

Now that you know a bit about objects, a whole new world is going to open up for you because JavaScript provides you with lots of objects (for doing math computations, manipulating strings, and creating dates and times, to name a few) that you can use in your own code. JavaScript also provides some really key objects that you need to write code for the browser (and we're going to take a look at one of those objects in the next chapter). For now, take a second to get acquainted with a few more of these objects; we'll touch on these throughout the rest of the book:



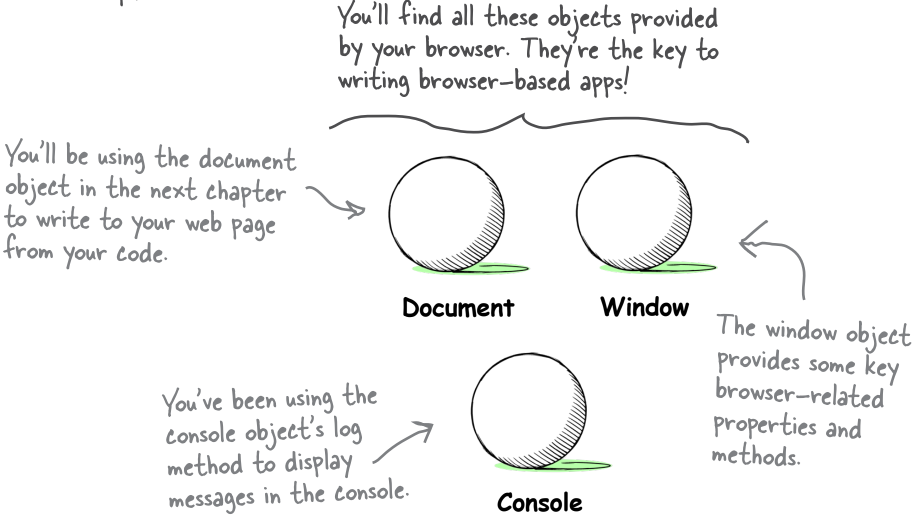
Use the Date object to manipulate dates and times.

You've already seen how to use the Math object to generate random numbers. It can do a lot more than that!

With JSON, you can exchange JavaScript objects with other applications.

This object lets you find patterns in strings.

All these objects are provided with JavaScript.



You'll find all these objects provided by your browser. They're the key to writing browser-based apps!

You'll be using the document object in the next chapter to write to your web page from your code.

The window object provides some key browser-related properties and methods.

You've been using the console object's log method to display messages in the console.

The Object Exposed

This week's interview: In Object's own words...

Head First: Welcome Object, it's been a fascinating chapter. It's a real head-spinner thinking about code as objects.

Object: Oh, well…we've only just begun.

Head First: How so?

Object: An object is a set of properties, right? Some of those properties are used to keep the state of the object, and some are actually functions—or rather, methods—that give an object behavior.

Head First: I'm with you so far. I hadn't actually thought about the methods being properties too, but I guess they are just another name and value, if you can call a function a value?

Object: Oh, you can! Believe me, you can. In fact, that's a huge insight, whether you realize it or not. Hold on to that thought; I'm guessing there's a lot in store for you on that topic.

Head First: But you were saying…

Object: So, you've looked at these objects with their properties and you've created lots of them, like a bunch of different types of cars.

Head First: Right…

Object: But it was very *ad hoc*. The real power comes when you can create a template of sorts, something that can basically stamp out uniform objects for you.

Head First: Oh, you mean objects that all have the same type?

Object: Sort of…as you'll see, the concept of type is an interesting one in JavaScript. But you're on the right track. You'll see that you have real power when you can start to write code that deals with objects of the same kind. Like, you could write code that deals with vehicles, and you wouldn't have to care if they were bicycles, cars, or buses. That's power.

Head First: It certainly sounds interesting. What else do we need to know to do that?

Object: Well, you have to understand objects a little better, and you need a way to create objects of the same kind.

Head First: We just did that, didn't we? All those cars?

Object: They're sort of the same kind by convention, because you happened to write code that creates cars that look alike. In other words, they have the same properties and methods.

Head First: Right, and in fact we talked a little about how we are replicating code across all those objects, which is not necessarily a good thing in terms of maintaining that code.

Object: The next step is to learn how to create objects that really are all guaranteed to be the same, and that make use of the same code—code that's all in one place. That's getting into how to design object-oriented code. And you're pretty much ready for that now that you know the basics.

Head First: I'm sure our readers are happy to hear that!

Object: But there are a few more things about objects to be aware of.

Head First: Oh?

Object: There are many objects already out there in the wild that you can use in your code.

Head First: Oh? I hadn't noticed, where?

Object: How about `console.log`. What do you think `console` is?

Head First: Based on this discussion, I'm guessing it's an object?

Object: BINGO. And `log`?

Head First: A property…err, a method?

Object: BINGO again. And what about `alert`?

Head First: I haven't a clue.

Object: It has to do with an object, but we'll save that for a bit later.

Head First: Well, you've certainly given us a lot to think about, Object, and I'm hoping you'll join us again.

Object: I'm sure we can make that work.

Head First: Great! Until next time then.

Crack the Code Challenge

In his quest for world domination, Dr. Evel has accidentally exposed an internal web page with the current passcode to his operation. With the passcode, we can finally get the upper hand. Of course, as soon as Dr. Evel discovered the page was live on the internet, he quickly took it down. Luckily, our agents made a record of the page. The only problem is, our agents don't know HTML or JavaScript. Can you help figure out the access code using the code below? Keep in mind, if you are wrong, it could be quite costly to Queen and Country.

TOP SECRET

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Here's the JavaScript.

Looks like this code is using a document object.

What passcode will you see in the alert?

Write your answer in the alert dialog box below.

Here's the HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

The above JavaScript code is being included here.

TOP SECRET
Looks like this code is using a document object.
What pass code will you see in the alert?
Here's the JavaScript
Write your answer in the alert dialog box below.
OK
Here's the HTML.
The above JavaScript code is being included here.

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

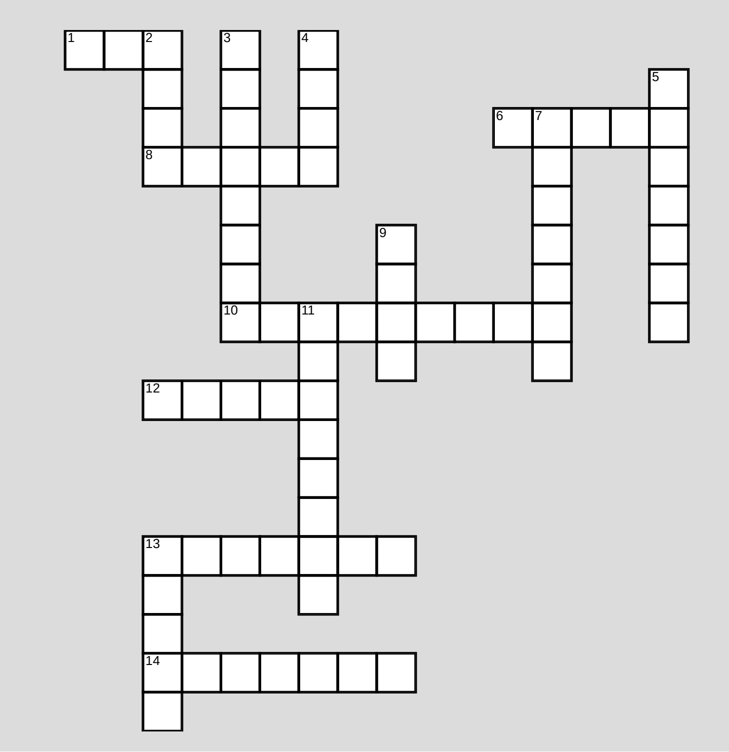
If you skipped the last page, go back and do the challenge. It is vitally important to Chapter 6!

Bullet Points

- An object is a collection of **properties**.
- To access a property, use **dot notation**: the name of the variable containing the object, then a period, then the name of the property.
- You can also access a property using **bracket notation**: the name of the variable containing the object, then the name of the property as a string in brackets.
- You can add new properties to an object at any time, by assigning a value to a new property name.
- You can also delete properties from objects, using the **delete** operator.
- Unlike variables that contain primitive values, like strings, numbers, and booleans, a variable can't actually contain an object. Instead, it contains a **reference** to an object. We say that objects are "reference variables."
- When you pass an object to a function, the function gets a copy of the reference to the object, not a copy of the object itself. So, if you change the value of one of the object's properties, it changes the value in the original object.
- Object properties can contain functions. When a function is in an object, we call it a **method**.
- You call a method by using **dot notation**: the object name, a period, and the property name of the method, followed by parentheses.
- A method is just like a function except that it is in an object.
- You can pass arguments to methods, just like you can to regular functions.
- When you call an object's method, the keyword **this** in the method refers to the object whose method you are calling.
- To access an object's properties in an object's method, use dot notation or bracket notation, with **this** in place of the object's name.
- In object-oriented programming, we think in terms of objects rather than procedures.
- An object has both **state** (properties) and **behavior** (methods). State can affect behavior, and behavior can affect state.
- Objects **encapsulate**, or hide, the complexity of the state and behavior of that object.
- A well-designed object has methods that abstract the details of how to get work done with the object, so you don't have to worry about it.
- Along with the objects you create, JavaScript has many built-in objects that you can use. We'll be using many of these built-in objects throughout the rest of the book.

JavaScript cross

How about a crossword object? It's got lots of clue properties that will help objects stick in your brain.



ACROSS

1. To access the properties of an object we use _______ notation.
6. Object references are passed by _______ to functions, just like primitive variables.
8. car and dog objects can have both _______ and behavior.
10. When you assign an object to a variable, the variable contains a _______ to the object.
12. The name and value of a property of an object are separated by a _______.
13. The method log is a property of the _______ object.
14. _______ can have local variables and parameters, just like regular functions can.

DOWN

2. The Fiat wouldn't start because we weren't using _______ to access the started property.
3. An object gets _______ with its methods.
4. We used a _______ property to represent the make of a car object.
5. this is a _______, not a regular variable.
7. The _______ method affects the state of the car object, by adding to the amount of fuel in the car.
9. We usually use one _______ for property names.
11. Shorthand notation for methods eliminates the colon and the _______ keyword.
13. Don't forget to use a _______ after each property value except the last one.

Solution on page 237

Sharpen your pencil Solution

From page 184

We've started making a table of property names and values for a car. Can you help complete it? Here's our solution.

Put your property names here.

And put the corresponding values over here.

```
{
  make        : "Chevy"      ,
  model       : "Bel Air"    ,
  year        : 1957         ,
  color       : "red"        ,
  passengers  : 2            ,
  convertible : false        ,
  mileage     : 1021         ,
  accessories : "Fuzzy Dice" ,
  whitewalls  : true
};
```

We're using strings, booleans, and numbers where appropriate.

Put your answers here. Feel free to expand the list to include your own properties.

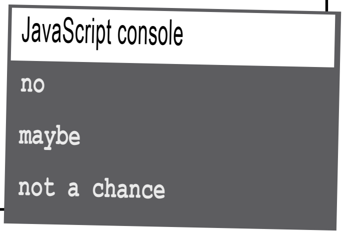
Sharpen your pencil Solution

From page 213

Use your new **this** skills to help us finish this code. Here's our solution.

```
              let eightBall = { index: 0,
                                advice: ["yes", "no", "maybe", "not a chance"],
                                shake: function() {
                                    this.index = this.index + 1;
                               if (this.index  >= this.advice.length) {
                                  this.index = 0;
                               }
                   },
                   look: function() {
                               return this.advice[this.index];
                   }

       };
       eightBall.shake();
       console.log(eightBall.look());
```

Repeat this sequence several times to test your code.



```
no
maybe
not a chance
```

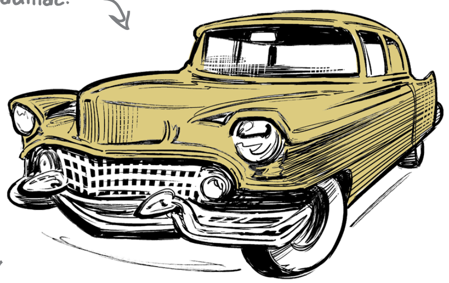
Exercise Solution

From page 186

You don't have to be stuck with just one object. The real power of objects (as you'll see soon enough) is having lots of objects and writing code that can operate on whatever object you give it. Try your hand at creating another object from scratch…another car object. Go ahead and work out the code for your second object. Here's our solution.

```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};
```

Here are the properties for the Cadillac.



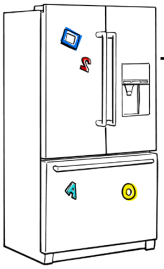
We'll call this a tan color.

This is a 1955 GM Cadillac.

It's not a convertible, and it can hold five passengers (it's got a nice big bucket seat in the back).

Its mileage is 12,892.

From page 191



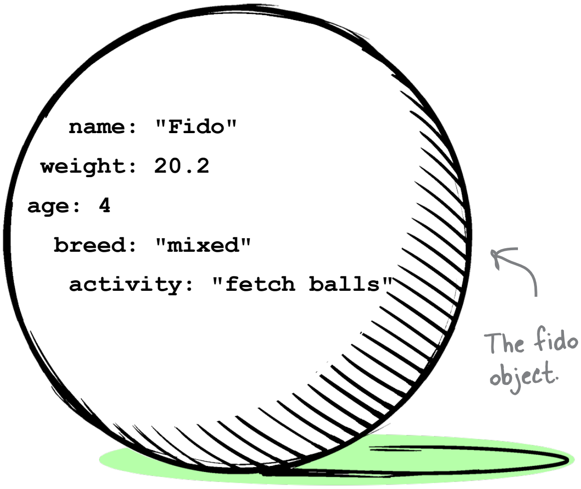
Object Magnets Solution

Practice your object creating and dot notation skills by completing the code below with the fridge magnets. Be careful, some extra magnets got mixed in! Here's our solution.

`,` `20.2` `Fido` `bark` `age`

Leftover magnets

```
let fido = {
    name: "Fido",
    weight: 20.2,
    age: 4,
    breed: "mixed",
    activity: "fetch balls"
};
let bark;
if (fido.weight > 20) {
    bark = "WOOF WOOF";
} else {
    bark = "woof woof";
}
let speak = fido.name + " says " + bark + " when he wants to " + fido.activity;
console.log(speak);
```

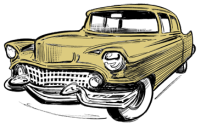

name: "Fido"
weight: 20.2
age: 4
breed: "mixed"
activity: "fetch balls"

The fido object.

From page 199

Sharpen your pencil Solution

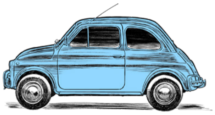
Your turn. Here are three more car objects; what is the result of passing each car to the prequal function? Work out the answers by hand, and then write the code to check your answers. Here's our solution.



```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};

prequal(cadi);
```

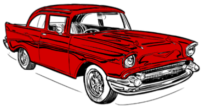
false

Write the value of prequal here.



```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

false



```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};

prequal(chevy);
```

true

From page 202

Sharpen your pencil Solution

You've been given a super-secret file and two functions that allow access to get and set the contents of the file, but only if you have the right password. The first function, getSecret, returns the contents of the file if the password is correct and logs each attempt to access the file. The second function, setSecret, updates the contents of the file and resets the access tracking back to 0. It's your job to fill in the blanks below to complete the JavaScript, and test your functions. Here's our solution.

```
function getSecret(file, secretPassword) {
    file.opened = file.opened + 1;
    if (secretPassword == file.password) {
        return file.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == file.password) {
        file.opened = 0;
        file.contents = secret;
    }
}

let superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
let secret = getSecret(superSecretFile, 2);
console.log(secret);

setSecret(superSecretFile, 2, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(superSecretFile, 2);
console.log(secret);
```

The superSecretFile object is passed into the getSecret function and gets the parameter name file. So, we need to make sure we use the object name, file, and dot notation to access the object's properties, like opened and password.

We can pass the supserSecretFile object to the getSecret and setSecret functions.

From page 211

BE the Browser Solution

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. Here's our solution.

```
let song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,
    play: function() {
        if (!this.playing) {
            this.playing = true;
            console.log("Playing "
                + this.name + " by " + this.artist);
        }
    },

    pause: function() {
        if (this.playing) {
            this.playing = false;
        }
    }
};
```

We were missing a this here.

And missing the playing property name here.

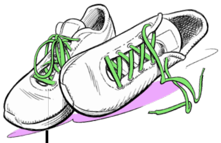
We need to use this to access both these properties, too.

Again here, we need this to access the playing property.

~~`this`~~ `song.play();`

~~`this`~~ `song.pause();`

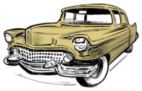
We don't use this outside of a method; we call an object using the object's variable name.



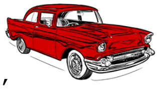
Exercise Solution

From page 215

It's time to get the whole fleet up and running. Add the drive method to each car object. When you've done that, add the code to start, drive, and stop each of them. Here's our solution.



```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```



```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

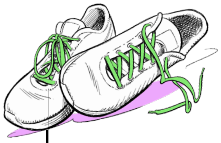
```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Make sure you add a comma after any new properties you add.

```
cadi.start();
cadi.drive();
cadi.stop();

chevy.start();
chevy.drive();
chevy.stop();

taxi.start();
taxi.drive();
taxi.stop();
```

We copied and pasted the code into each object, so every car has the same properties and methods.

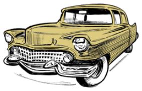
Now we can start, drive, and stop each of the cars, using the same method names.



Exercise Solution

From page 218

Go ahead and update your code for the cadi, chevy, and taxi objects to use the method shorthand. Here's our solution.



```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

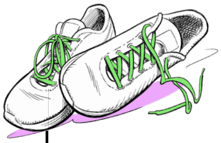
```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Make sure you add a comma after any new properties you add.

```
cadi.start();
cadi.drive();
cadi.stop();

chevy.start();
chevy.drive();
chevy.stop();

taxi.start();
taxi.drive();
taxi.stop();
```

We copied and pasted the code into each object, so every car has the same properties and methods.

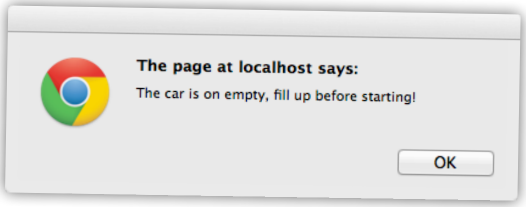
Now we can start, drive, and stop each of the cars, using the same method names.



Exercise Solution

From page 221

We still have some more work to do to fully integrate the fuel property into the car. For instance, should you really be able to start the car if there's no fuel? Help us integrate the fuel property into this code by checking the fuel level before the car is started. If there's no fuel and the start method is called, let the driver know with a handy alert like **`"The car is on empty, fill up before starting!"`** Rewrite the start method below, and then add it to your code and test it. Here's our solution.

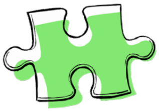


The page at localhost says:

The car is on empty, fill up before starting!

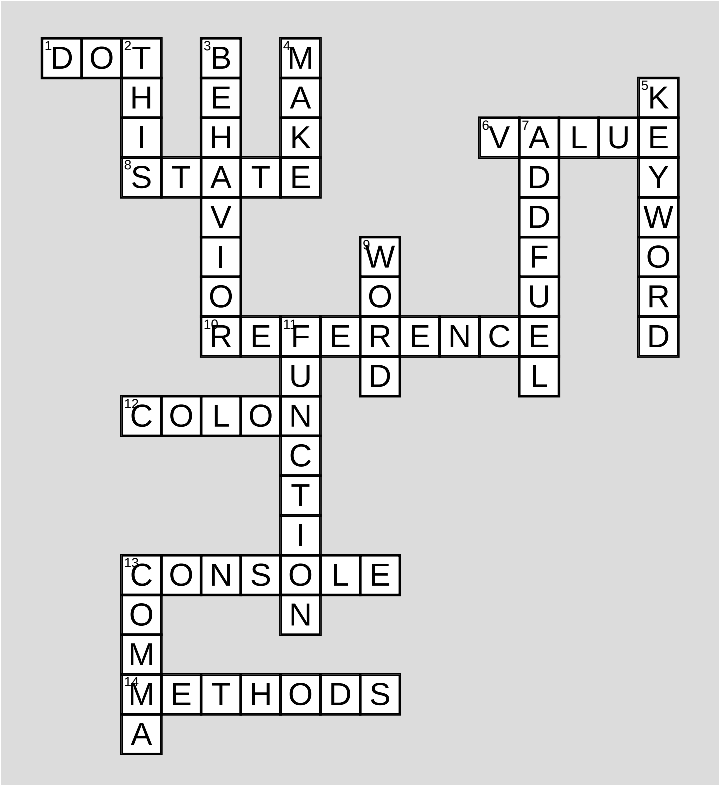
```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    fuel: 0,
    started: false,

    start() {
        if (this.fuel == 0) {
            alert("The car is on empty, fill up before starting!");
        } else {
            this.started = true;
        }
    },

    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
                this.fuel = this.fuel - 1;
            } else {
                alert("Uh-oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

From page 227



JavaScript cross Solution

How about a crossword object? It's got lots of clue properties that will help objects stick in your brain. Here's our solution.


1 DOT
2 THIS
3 BEHAVIOR
4 MAKE
5 KEYWORD
6 VALUE
7 ADDFUEL
8 STATE
9 WORD
10 REFERENCE
11 FUNCTION
12 COLON
13 CONSOLE
13 COMMA
14 METHODS

6 interacting with your web page

Getting to know the DOM

You’ve come a long way with JavaScript. In fact, you’ve evolved from a newbie to a scripter to, well, a **programmer**. But, there's something missing. To really begin leveraging your JavaScript skills, you need to know how to interact with the web page your code lives in. Only by doing that are you going to be able to write pages that are **dynamic**, pages that react, that respond, that update themselves after they’ve been loaded. So how do you interact with the page? By using the **DOM**, otherwise known as the **document object model**. In this chapter, we’re going to break down the DOM and show you how you can use it, along with JavaScript, to teach your pages a few new tricks.

In the last chapter, we left you with a little challenge...the "crack the code challenge"

TOP SECRET

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Looks like this code is using a document object.

Here's the JavaScript

What pass code will you see in the alert?

Write your answer in the alert dialog box below.

Here's the HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

The above JavaScript code is being included here.

You were given some HTML with code in an external file, captured from Dr. Evel's website, that looked like this:

Here's the HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

Notice that each paragraph is identified by an id.

Here's the JavaScript...

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```

document is a global object in the browser.

And getElementById is a method.

Make sure you get the case right on the letters in the method name getElementById; otherwise, it won't work!

And look, we have dot notation; this looks like an object with an innerHTML property.

And you needed to figure out Dr. Evel's passcode using your deductive powers on this code.

So what does the code do?

Let's walk through this code to see how Dr. Evel is generating his passcodes. After we break down each step, you'll start to see how this all works:

You'll learn all about document and element objects in this chapter.

1. First, the code sets the variable `access` to the result of calling the `document` object's `getElementById` method and passing it "code9". What gets returned is an element object.

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Get the element that has an id of "code9". That would be this element...

```
<p id="code9">My watch stopped at</p>
```

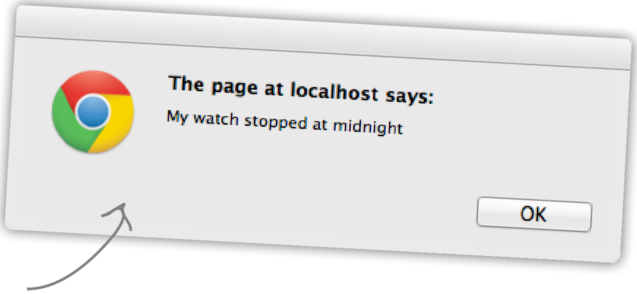
2. Next, we take that element (that is, the element with the id "code9") and we use its `innerHTML` property to get its content, which we assign to the variable `code`.

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```

The element with id "code9" is a paragraph element, and that element's content (or rather, its "innerHTML") is the text "My watch stopped at".

3. Dr. Evel's code adds the string " midnight" to the end of the string contained in `code`, which is "My watch stopped at". Then, the page creates an alert with the passcode contained in the variable `code`.

```
let access =
  document.getElementById("code9");
let code = access.innerHTML;
code = code + " midnight";
alert(code);
```



The page at localhost says:

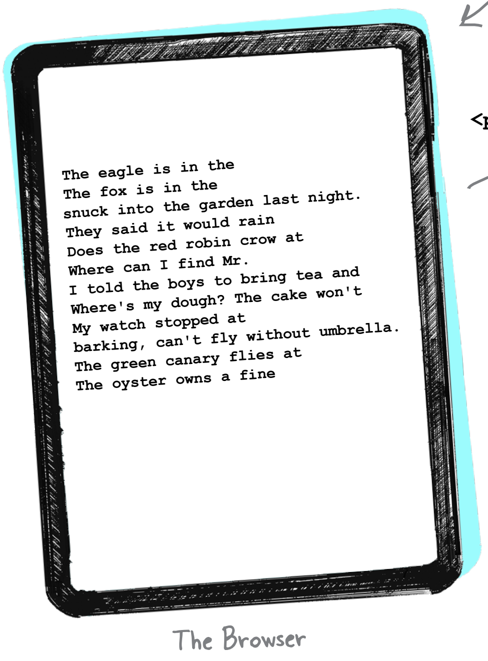
My watch stopped at midnight

We add " midnight" to "My watch stopped at" to get "My watch stopped at midnight" and then put up an alert to display this code.

A quick recap

So, what did we just do? Well, we had some JavaScript that reached into the page (otherwise known as the *document*), grabbed an element (the one with the id equal to "code9"), took that element's content (which is "My watch stopped at"), slapped a " midnight" on the end, and then displayed the result as a passcode.

① Dr. Evel's page contains all the possible passcodes, each in a paragraph element labeled with an HTML id.

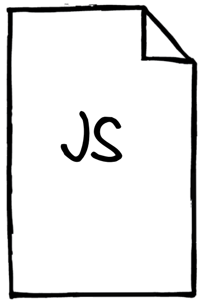


```
The eagle is in the
The fox is in the
snuck into the garden last night.
They said it would rain
Does the red robin crow at
Where can I find Mr.
I told the boys to bring tea and
Where's my dough? The cake won't
My watch stopped at
barking, can't fly without umbrella.
The green canary flies at
The oyster owns a fine
```

The Browser

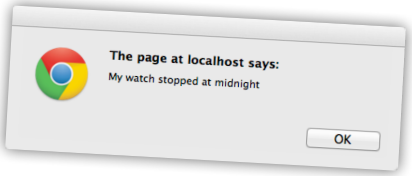
② Behind the scenes, JavaScript grabs the element with id="code9".

`<p id="code9">My watch stopped at</p>`



`"My watch stopped at" + " midnight"`

③ It then takes the content of that element, adds " midnight" to it, and displays an alert.

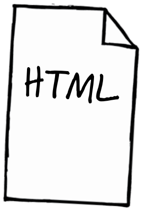


The page at localhost says:
My watch stopped at midnight

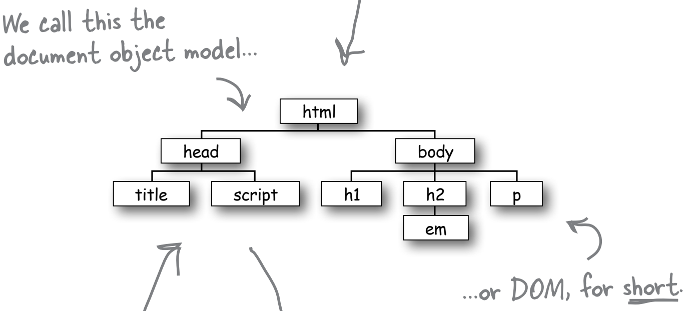
Now, more power to Dr. Evel and his JavaScript skills, and we wish him the best in his security schemes, but what is important here is to notice that the web page is a living, breathing *data structure* that your JavaScript can interact with—you can access and read the contents of the elements in your page. You can also go the other way, and use JavaScript to change the content or structure of your page. To do all that, let's step back for a moment and think about how JavaScript and HTML work together.

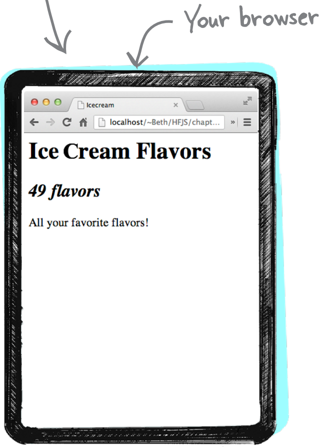
How JavaScript really interacts with your page

JavaScript and HTML are *two different things*. HTML is markup, and JavaScript is code. So how do they interact? It all happens through a representation of your page, called the *document object model*, or the DOM for short. Where does the DOM come from? It's created when the browser loads your page. Here's how:



① When you load a page into the browser, not only does the browser parse the HTML and then render it to the display, it also creates a set of objects that represent the HTML. These objects are stored in the DOM.


We call this the document object model...
html
head
body
title
script
h1
h2
p
em
...or DOM, for short.

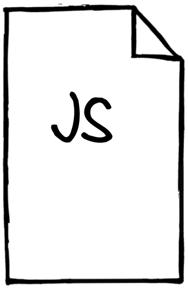

Your browser
Icecream
localhost/~Beth/HFJS/chapt...

Ice Cream Flavors

49 flavors

All your favorite flavors!

② Your JavaScript can interact with the DOM to get access to the elements and their contents. JavaScript can also use the DOM to create or remove elements.



③ When JavaScript modifies the DOM, the browser updates the page dynamically, so you see new content on your page.

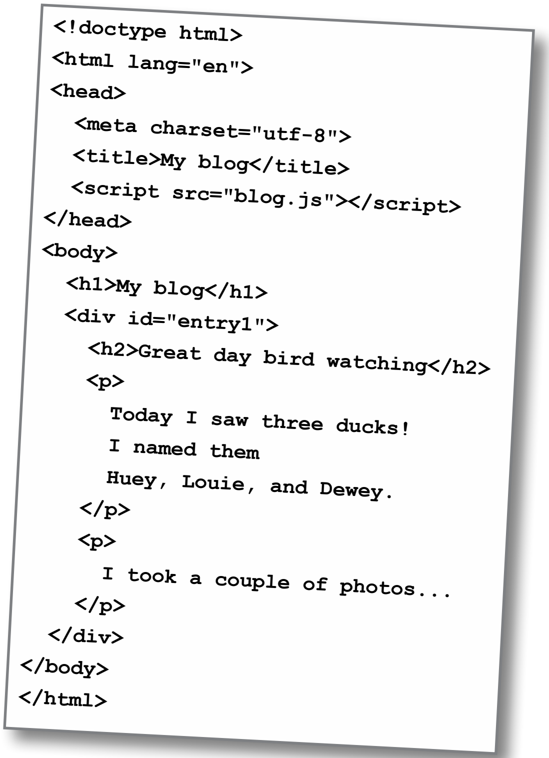
How to bake your very own DOM

Let's take some HTML and create a DOM for it. Here's a simple recipe for doing that:

Ingredients

One well-formed HTML page

One modern web browser, preheated and ready to go



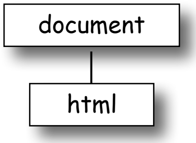
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My blog</title>
  <script src="blog.js"></script>
</head>
<body>
  <h1>My blog</h1>
  <div id="entry1">
    <h2>Great day bird watching</h2>
    <p>
      Today I saw three ducks!
      I named them
      Huey, Louie, and Dewey.
    </p>
    <p>
      I took a couple of photos...
    </p>
  </div>
</body>
</html>
```

We've already fully baked this DOM for you. See the finished DOM on the next page.

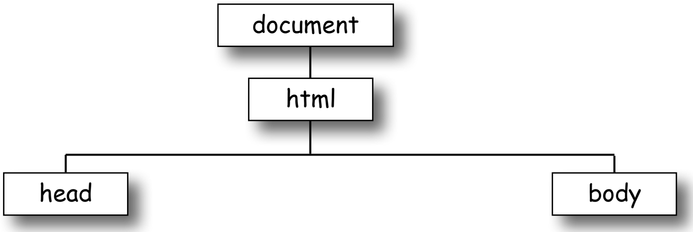
Instructions

1. Start by creating a document node at the top.

2. Next, take the top-level element of your HTML page, in our case the <html> element, call it the current element, and add it as a child of the document.



3. For each element nested in the current element, add that element as a child of the current element in the DOM.

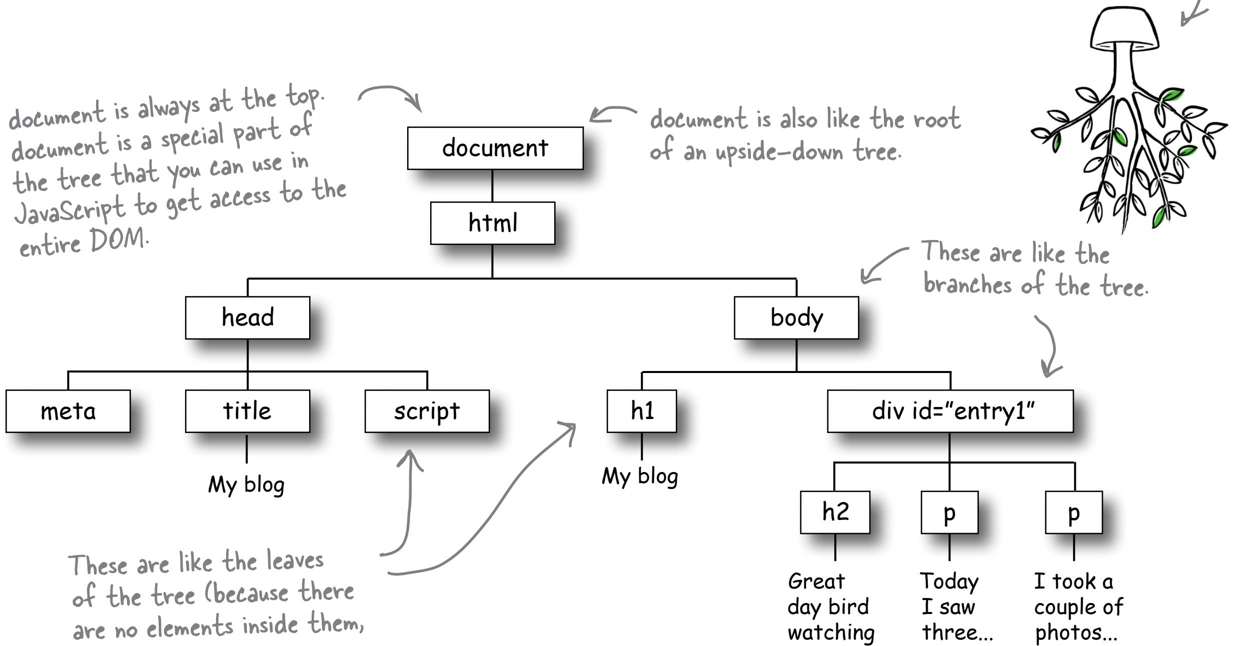


4. Return to (3) for each element you just added, and repeat until you are out of elements.

A first taste of the DOM

If you follow the recipe for creating a DOM, you'll end up with a structure like the one below. Every DOM has a `document` object at the top and then a tree complete with branches and leaf nodes for each element in the HTML markup. Let's take a closer look.

We compare this structure to a tree because a "tree" is a data structure that comes from computer science, and because it looks like an upside-down tree, with the root at the top and the leaves at the bottom.


document
html
head
body
meta
title
script
h1
div id="entry1"
My blog
My blog
h2
p
p
Great day bird watching
Today I saw three...
I took a couple of photos...

document is always at the top. document is a special part of the tree that you can use in JavaScript to get access to the entire DOM.

document is also like the root of an upside-down tree.

These are like the branches of the tree.

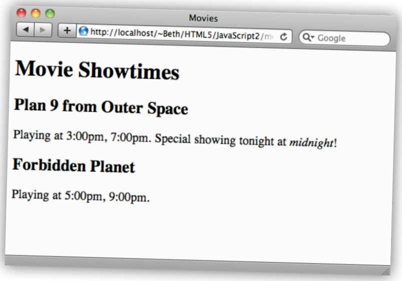
These are like the leaves of the tree (because there are no elements inside them, just text).

The DOM includes the content of the page as well as the elements. (We don't always show all the text content when we draw the DOM, but it's there.)

Now we have a DOM we can examine or alter in any way we want.

BE the Browser

Your job is to act like you're the browser. You need to parse the HTML and build your very own DOM from it. Go ahead and parse the following HTML, and draw your DOM below. We've already started it for you.


Movies
http://localhost/~Beth/HTML5/JavaScript2/m
Google

Movie Showtimes

Plan 9 from Outer Space

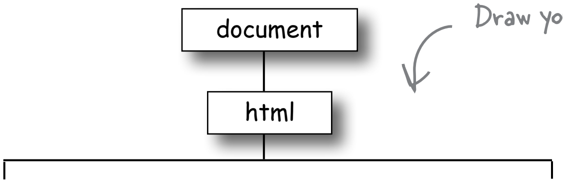
Playing at 3:00pm, 7:00pm. Special showing tonight at *midnight*!

Forbidden Planet

Playing at 5:00pm, 9:00pm.

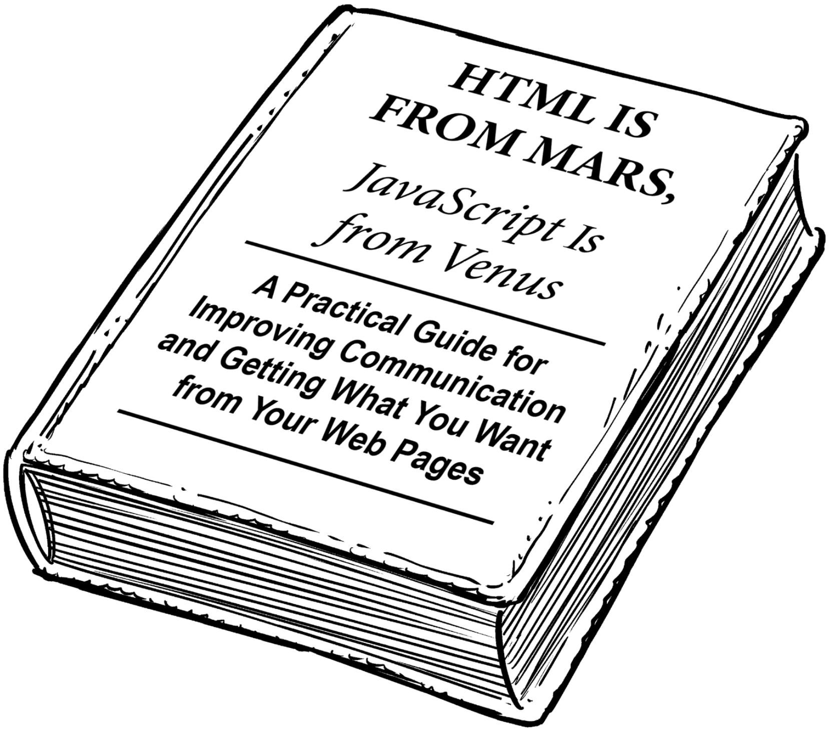
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Movies</title>
  </head>
  <body>
    <h1>Movie Showtimes</h1>
    <h2 id="movie1">Plan 9 from Outer Space</h2>
    <p>Playing at 3:00pm, 7:00pm.
      <span>
        Special showing tonight at <em>midnight</em>!
      </span>
    </p>
    <h2 id="movie2">Forbidden Planet</h2>
    <p>Playing at 5:00pm, 9:00pm.</p>
  </body>
</html>
```

Check your answer at the end of the chapter before you go on.


document
html

Draw your DOM here.

Solution on page 271

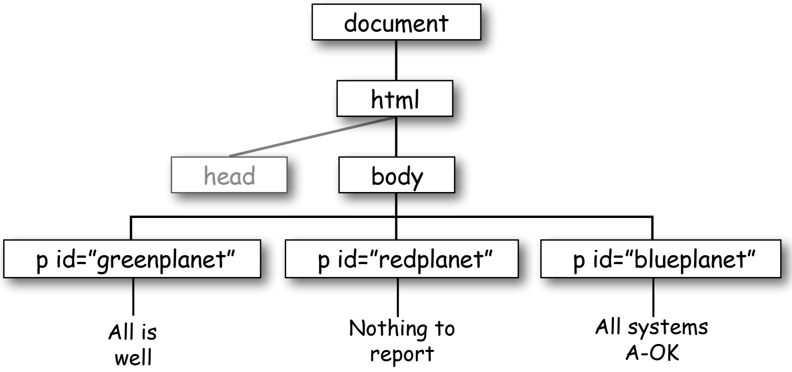

HTML IS
FROM MARS,
JavaScript Is
from Venus
A Practical Guide for
Improving Communication
and Getting What You Want
from Your Web Pages

Or, how two totally different technologies hooked up.

HTML and JavaScript are from different planets for sure. The proof? HTML's DNA is made of declarative markup that allows you to describe a set of nested elements that make up your pages. JavaScript, on the other hand, is made of pure algorithmic genetic material, meant for describing computations.

Are they so far apart they can't even communicate? Of course not, because they have something in common: the DOM. Through the DOM, JavaScript can communicate with your page, and vice versa. There are a few ways to make this happen, but for now let's concentrate on one—it's a little wormhole of sorts that allows JavaScript to get access to any element in your page. That wormhole is *getElementById*.

Let's start with a DOM. Here's a simple DOM; it's got a few HTML paragraphs, each with an id identifying it as the green, red, or blue planet. Each paragraph has some text as well. Of course, there's a <head> element too, but we've left the details out to keep things simpler.

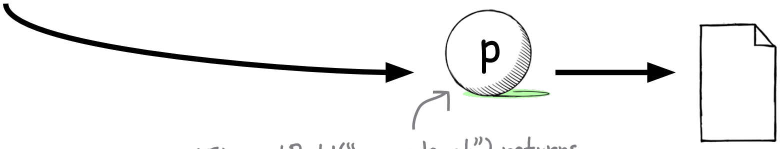


Now, let's use JavaScript to make things more interesting. Let's say we want to change the green planet's text from "All is well" to "Red Alert: hit by phaser fire!" Down the road you might want to do something like this based on a user's actions, or even based on data from a web service. We'll get to all that; for now, let's just get the green planet's text updated. To do that, we need to get the element with the id "greenplanet". Here's some code that does that:

The document represents the entire page in your browser and contains the complete DOM, so we can ask it to do things like find an element with a specific id.

Here, we're asking the document to get us an element by finding the element that matches the given id.

```
document.getElementById("greenplanet");
```



getElementById("greenplanet") returns the paragraph element corresponding to "greenplanet"...

...and then the JavaScript code can do all sorts of interesting things with it.

Once getElementById gives you an element, you're ready do something with it. (like change its text to "Red Alert: hit by phaser fire!"). To do that, we typically assign the element to a variable so we can refer to the element throughout our code. Let's do that and then change the text:

We're assigning the element to a variable named planet.

Here's our call to getElementById, which seeks out the "greenplanet" element and returns it.

```
let planet = document.getElementById("greenplanet");
```

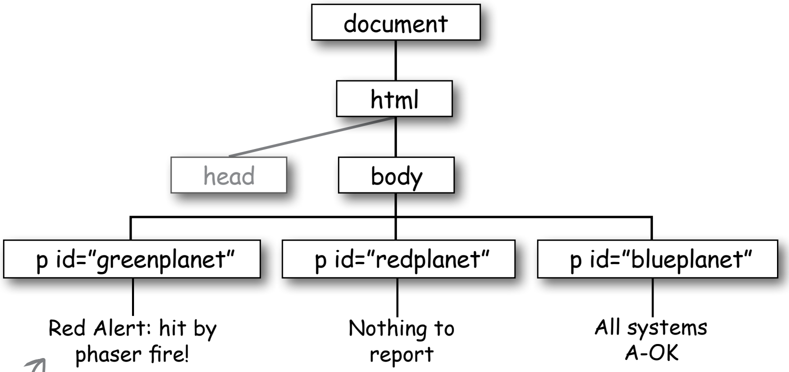
And in our code we can now just use the variable planet to refer to our element.

```
planet.innerHTML = "Red Alert: hit by phaser fire!";
```

We can use the innerHTML property of our planet element to change the content of the element.

We'll talk more about properties of elements shortly...

We change the content of the greenplanet element to our new text, which results in the DOM (and our page) being updated with the new text.


document
html
head
body
p id="greenplanet"
p id="redplanet"
p id="blueplanet"
Red Alert: hit by phaser fire!
Nothing to report
All systems A-OK

Any changes to the DOM are reflected in the browser's rendering of the page, so you'll see the paragraph change to contain the new content!

Getting an element with getElementById

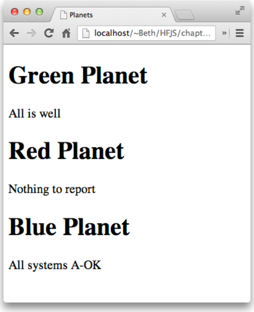
So, what did we just do? Let's step through it in a little more detail. We're using the `document` object to get access to the DOM from our code. The `document` object is a built-in object that comes with a bunch of properties and methods, including `getElementById`, which we can use to grab an element from the DOM. The `getElementById` method takes an id and returns the element that has that id. In the past, you've probably used ids to select and style elements with CSS. But here, what we're doing is using an id to grab an element—the `<p>` element with the id "greenplanet"—from the DOM.

Once we have the right element, we can modify it. We'll get to that in just a moment; for now, let's focus on how `getElementById` works by tracing through these steps:

Follow steps 1, 2, and 3.

1

Browser here. I'm reading the page and creating a DOM of it.



Planets

localhost/~Beth/HFJS/chapt...

Green Planet

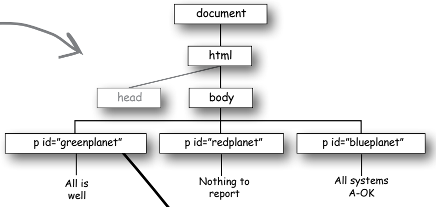
All is well

Red Planet

Nothing to report

Blue Planet

All systems A-OK



JavaScript code here. I'm looking for an element in the DOM with an id of "greenplanet".

We're using document to get access to the DOM.

2

```
let planet = document.getElementById("greenplanet");
```

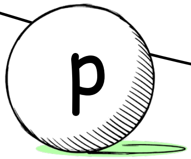
We'll assign the element that is returned to the planet variable for later use.

Here's our call to getElementById.

We're looking for the element with an id of "greenplanet".

You found me! I'm the <p> element with the "greenplanet" id. Just tell me what you need me to do.

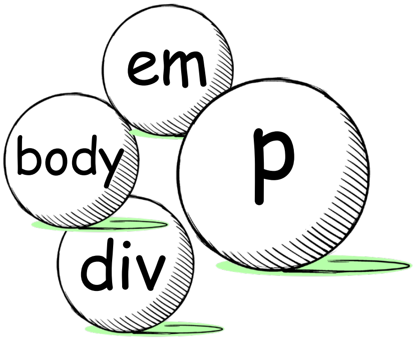
3



What, exactly, am I getting from the DOM?

When you grab an element from the DOM using `getElementById`, what you get is an *element object*, which you can use to read, change, or replace the element's content and attributes. And here's the magic: when you change an element, *you change what is displayed in your page as well.*

But first things first. Let's take another look at the element object we just grabbed from the DOM. We know that this element object represents the `<p>` element in our page that has the id "greenplanet" and that the text content in the element is "All is well". Just like other kinds of JavaScript objects, an element object has properties and methods. In the case of an element object, we can use these properties and methods to read and change the element. Here are a few things you can do with element objects:



Get the content (text or HTML).

Change the content.

Read an attribute.

Add an attribute.

Change an attribute.

Remove an attribute.

Things you can do with an element object.

What we want to do with our `<p>` element—which, remember, is the `<p>` element with the id "greenplanet"—is change the content "All is well" to "Red Alert: hit by phaser fire!" We've got the element object stashed in the `planet` variable in our code; let's use that to modify one of its properties, `innerHTML`:

The planet variable contains an element object—the element object that is the "greenplanet" <p> element.

```
let planet = document.getElementById("greenplanet");

planet.innerHTML = "Red Alert: hit by phaser fire!";
```

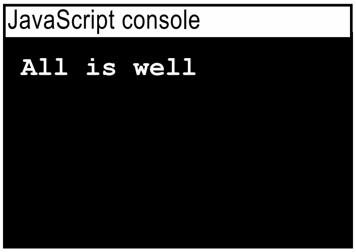
We can use the innerHTML property of the element object to change the content of the element!

Finding your inner HTML

The `innerHTML` property is an important property that we can use to read or replace the content of an element. If you look at the value of `innerHTML`, you'll see the content contained *within* the element, not including the HTML element tags. The "withIN" is why it's called "inner" HTML. Let's try a little experiment. We'll try displaying the content of the `planet` element object in the console by logging the `innerHTML` property. Here's what we get:

```
let planet = document.getElementById("greenplanet");
console.log(planet.innerHTML);
```

We're just passing the planet.innerHTML property to console.log to log to the console.



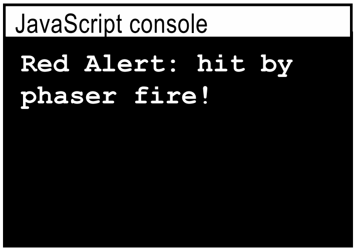
```
All is well
```

The content of the innerHTML property is just a string, so it displays like any other string in the console.

Now let's try changing the value of the `innerHTML` property. When we do this, we're changing the content of the "greenplanet" <p> element in the page, so you'll see your page change too!

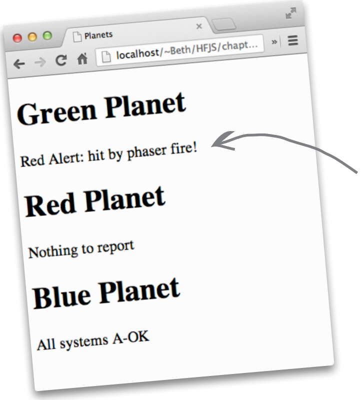
```
let planet = document.getElementById("greenplanet");
planet.innerHTML = "Red Alert: hit by phaser fire!";
console.log(planet.innerHTML);
```

Now we're changing the content of the element by setting its innerHTML property to the string "Red Alert: hit by phaser fire!"



```
Red Alert: hit by
phaser fire!
```

So when we log the value of the innerHTML property to the console, we see the new value.



Green Planet

Red Alert: hit by phaser fire!

Red Planet

Nothing to report

Blue Planet

All systems A-OK

And the web page changes too!

A Quick Refresher

PUMPKIN
COFFEE

Hey, sit down; take a quick break. You might be saying to yourself, "Wait, I remember something about ids and classes, but I don't remember the specifics, and don't they have something to do with CSS anyway?" No, problem; let's just have a quick refresher, get some context, and we'll have you back on your way in no time...

With HTML, ids give us a way to uniquely identify an element. Once an element has a unique id, we can use that to select it with CSS for styling. And, as you've seen, we can get an element by its id in JavaScript as well.

Let's look at an example:

```
<div id="menu">
  ...
</div>
```

We're giving this <div> a unique id of "menu". It should be the only element in our page with the id "menu".

We can select this unique id with CSS to style it, like this:

div#menu is an id selector.

```
div#menu {
    background-color: #aaa;
}
```

div#menu selects the <div> with the id "menu", so we can apply style to that element, and only that element.

And we can access this element through its id in JavaScript too:

```
let myMenu = document.getElementById("menu");
```

Don't forget, there's another way to label your elements: with classes. Classes give us a way to label a set of elements, like this:

```
<h3 class="drink">Strawberry Blast</h3>
<h3 class="drink">Lemon Ice</h3>
```

Both <h3> elements are in the class "drink". A class is like a group; you can have multiple elements in the same group.

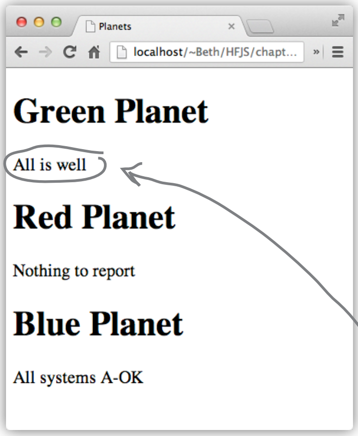
And we can select elements by classes too, in both CSS and JavaScript. We'll see how to make use of classes with JavaScript in a bit. If you'd like a more in-depth refresher, check out Chapter 7 of *Head First HTML and CSS,* or your favorite HTML and CSS reference guide.

What happens when you change the DOM

So, what exactly happens when you change the content of an element using `innerHTML`? What you're doing is changing the actual content of your web page, on the fly. And when you change the content in the DOM, you'll see that change immediately in your web page, too.

Before...

The web page you see and the DOM behind the scenes before you change the content with innerHTML...


Planets
localhost/~Beth/HFJS/chapt...

Green Planet

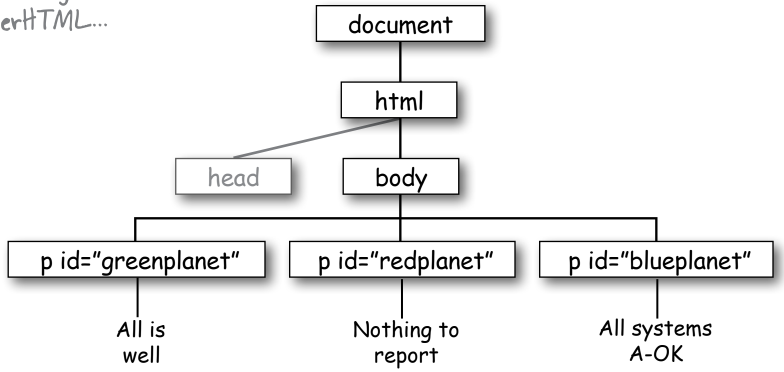
All is well

Red Planet

Nothing to report

Blue Planet

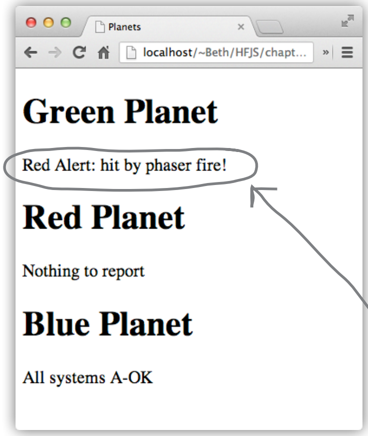
All systems A-OK


document
html
head
body
p id="greenplanet"
p id="redplanet"
p id="blueplanet"
All is well
Nothing to report
All systems A-OK

This is the element whose content we're going to change.

...and after.

...and the web page you see and the DOM behind the scenes after you change the content with innerHTML.


Planets
localhost/~Beth/HFJS/chapt...

Green Planet

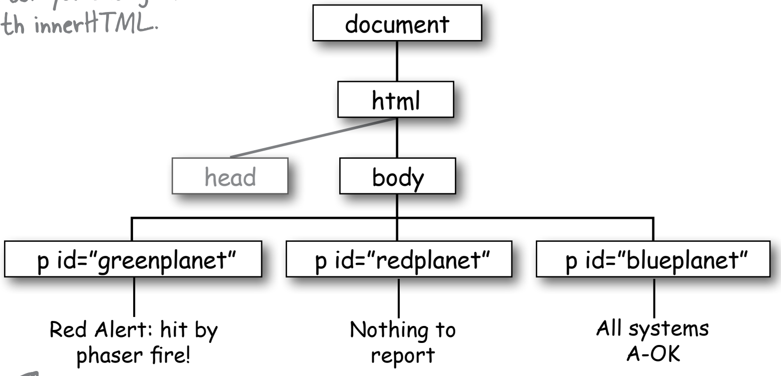
Red Alert: hit by phaser fire!

Red Planet

Nothing to report

Blue Planet

All systems A-OK


document
html
head
body
p id="greenplanet"
p id="redplanet"
p id="blueplanet"
Red Alert: hit by phaser fire!
Nothing to report
All systems A-OK

Any changes to the DOM are reflected in the browser's rendering of the page, so you'll see the paragraph change to contain the new content.

there are no Dumb Questions

Q: What happens if I pass an id that doesn't exist to document.getElementById?

A: If you try to get an element from the DOM by id, and that id doesn't exist in an element, then the call to getElementById returns a null value. Testing for null is a good idea when you use getElementById, to ensure that the element is there before you try to access its properties. We'll talk more about null in the next chapter.

Q: Can I use document.getElementById to get elements by class as well—for instance, if I have a bunch of elements in the class "planets"?

A: No, but you're thinking along the right lines. You can only use getElementById with an id, but there is another DOM method named getElementsByClassName that you can use to get elements by class name. With this method, what you get back is a collection of elements that belong to the class (because multiple elements can be in the same class). Another method that returns a collection of elements is getElementsByTagName, which returns all elements that match the tag name you specify. We'll look at getElementsByTagName a little later in the book and see how to handle the collection of elements it returns.

Q: Are there any other ways I can select elements in the DOM?

A: Yes, you can select elements with a CSS selector. That's a bit beyond the scope of this book, but if you know how to use CSS selectors, you can use document.querySelector, which returns the first element that matches the selector. You can also use document.querySelectorAll to return a collection of elements that match the selector. For querySelector and querySelectorAll, null is returned if no matching elements exist. These are both very powerful ways to locate elements in the DOM.

Q: What exactly is an element object, anyway?

A: Great question. An element object is the browser's internal representation of what you type into your HTML file, like <p>some text</p>. When the browser loads and parses your HTML file, it creates an element object for every element in your page and adds all those element objects to the DOM. So, the DOM is really just a big tree of element objects. And keep in mind that, just like other objects, element objects can have properties, like innerHTML, and methods, too. We'll explore a few more of the properties and methods of element objects later in the book.

Q: I would have expected a property named "content" or maybe "html" in the element object. Why is it called innerHTML instead?

A: We agree, it's kind of a weird name. The innerHTML property represents all the content contained in your element, including other nested elements (like how a paragraph might include <em> and <img> elements in addition to the text in the paragraph). In other words, it's the HTML that's "INside" your element. Is there an outerHTML property? Yes! And that property gets you all the HTML inside the element, as well as the element itself. In practice you won't see outerHTML used very often, but you will see innerHTML used frequently to update the content of elements.

Q: So, by assigning something to innerHTML I can replace the content of any element with something else. What if I used innerHTML to change, say, the <body> element's content?

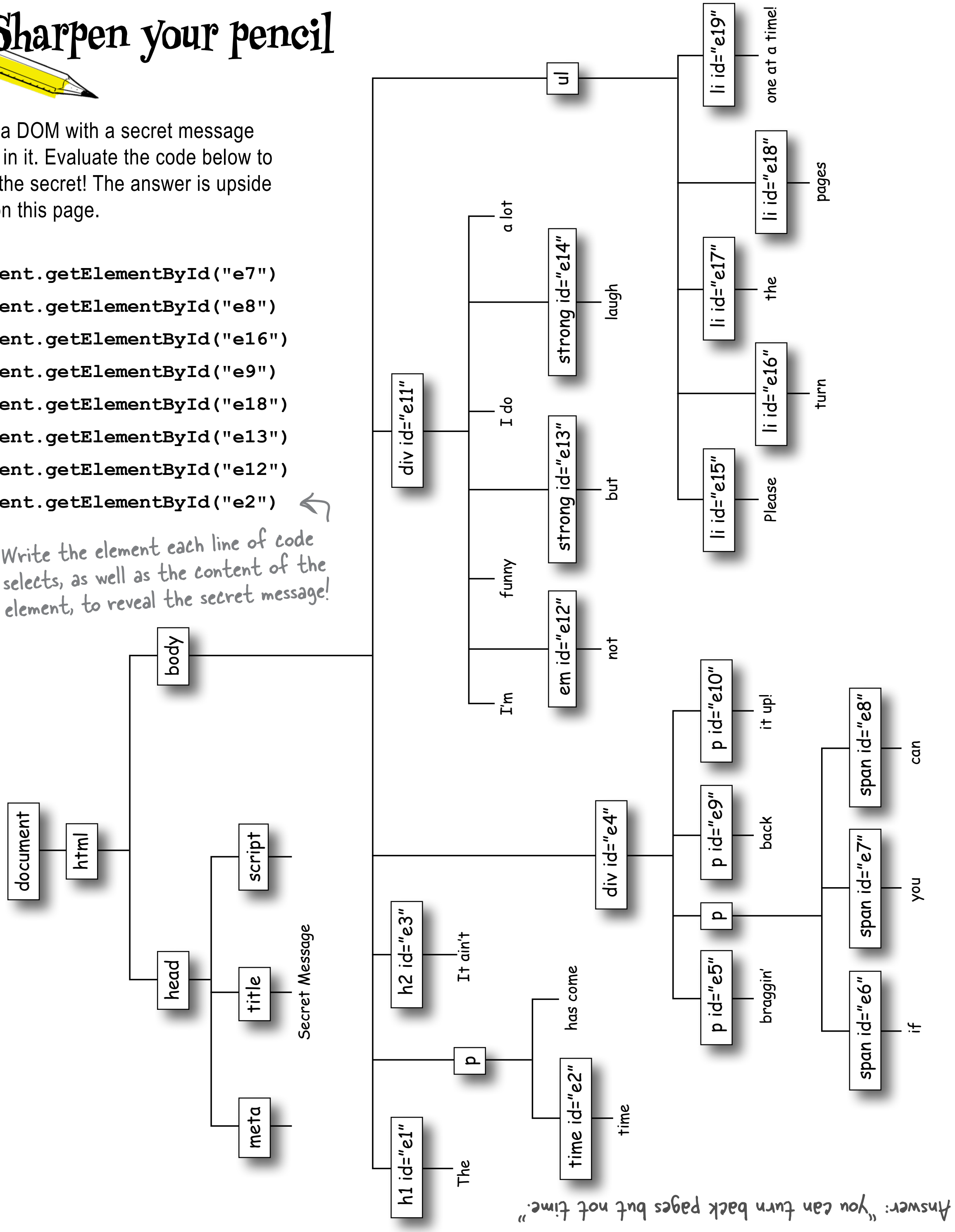
A: Right, innerHTML gives you a convenient way to replace the content of an element. And, yes, you could use it to replace the content of the <body> element, which would result in your entire page being replaced with something new.

Sharpen your pencil

Here's a DOM with a secret message hidden in it. Evaluate the code below to reveal the secret! The answer is upside down on this page.

```
document.getElementById("e7")
document.getElementById("e8")
document.getElementById("e16")
document.getElementById("e9")
document.getElementById("e18")
document.getElementById("e13")
document.getElementById("e12")
document.getElementById("e2")
```

Write the element each line of code selects, as well as the content of the element, to reveal the secret message!


document
html
head
meta
title
Secret Message
script
body
h1 id="e1"
The
p
time id="e2"
time
has come
h2 id="e3"
It ain't
div id="e4"
p id="e5"
braggin'
p
span id="e6"
if
span id="e7"
you
span id="e8"
can
p id="e9"
back
p id="e10"
it up!
div id="e11"
I'm
em id="e12"
not
funny
strong id="e13"
but
I do
strong id="e14"
laugh
a lot
ul
li id="e15"
Please
li id="e16"
turn
li id="e17"
the
li id="e18"
pages
li id="e19"
one at a time!

Answer: "you can turn back pages but not time."

A test drive around the planets

Okay, you know how to use `document.getElementById` to get access to an element, and how to use `innerHTML` to change the content of that element. Let's do it for real, now.

Here's the HTML for the planets. We've got a `<script>` element in the `<head>` where we'll put the code, and three paragraphs for the green, red, and blue planets. If you haven't already, go ahead and type in the HTML and the JavaScript to update the DOM:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <script>
    let planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

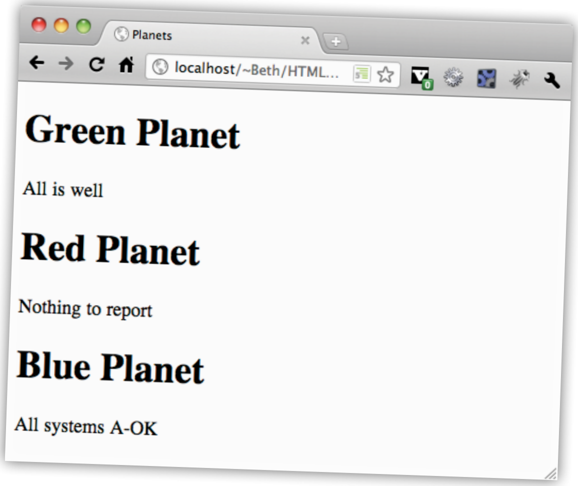
Here's our <script> element with the code.

Just like you saw before, we're getting the <p> element with the id "greenplanet" and changing its content.

Here's the <p> element you're going to change with JavaScript.

After you've got it typed in, go ahead and load the page into your browser and see the DOM magic happen on the green planet.

UH-OH! Houston, we've got a problem: the green planet still shows "All is well". What's wrong?



Green Planet

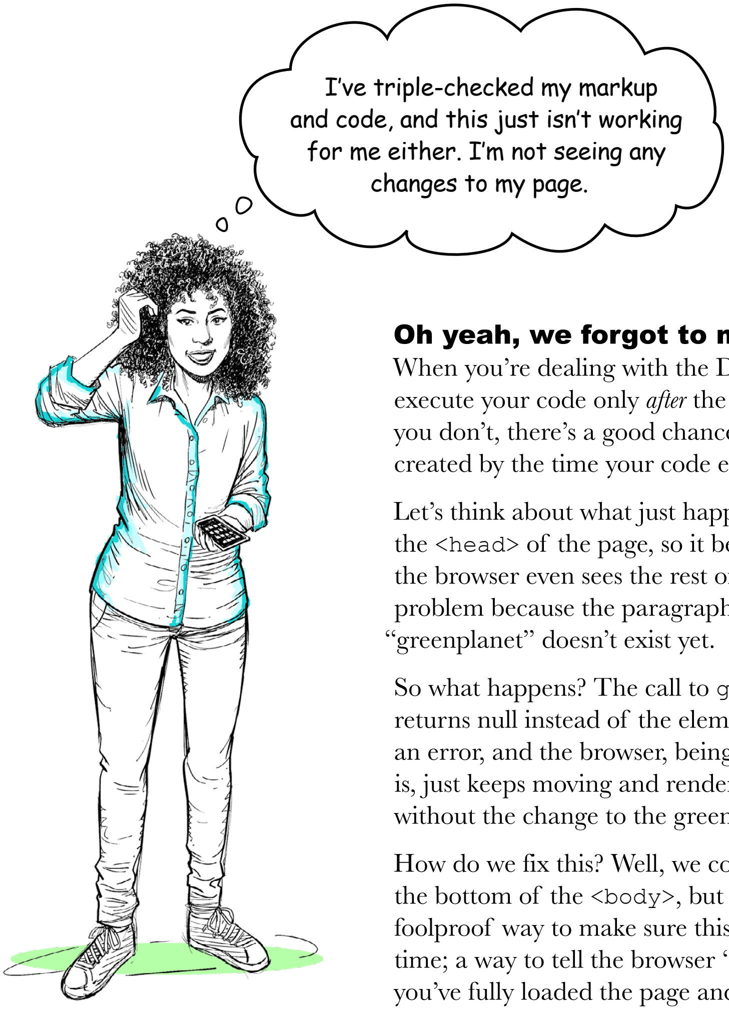
All is well

Red Planet

Nothing to report

Blue Planet

All systems A-OK


I've triple-checked my markup and code, and this just isn't working for me either. I'm not seeing any changes to my page.

Oh yeah, we forgot to mention one thing.

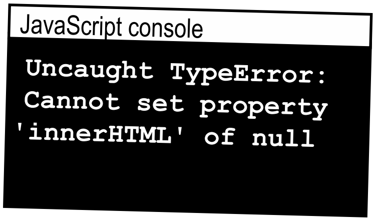
When you're dealing with the DOM, it's important to execute your code only *after* the page is *fully loaded*. If you don't, there's a good chance the DOM won't be created by the time your code executes.

Let's think about what just happened: we put code in the `<head>` of the page, so it begins executing before the browser even sees the rest of the page. That's a big problem because the paragraph element with an id of "greenplanet" doesn't exist yet.

So what happens? The call to `getElementById` returns null instead of the element we want, causing an error, and the browser, being the good sport that it is, just keeps moving and renders the page anyway, but without the change to the greenplanet content.

Check out your console when this page loads—you'll see the error in most browsers. The console tool is good for debugging.

How do we fix this? Well, we could move the code to the bottom of the `<body>`, but there's actually a more foolproof way to make sure this code runs at the right time; a way to tell the browser "run my code after you've fully loaded the page and created the DOM." Let's look at that next.


JavaScript console

```
Uncaught TypeError:
Cannot set property
'innerHTML' of null
```

Don't even think about running my code until the page is fully loaded!

Ah, but how? Besides moving the code to the bottom of the body, there's another—and, one might argue, cleaner—way to do it: *with code.*

Here's how it works: first create a function that has the code you'd like executed *once the page is fully loaded*, then take the window object and assign the function to its `onload` property.

What does that do? The window object will call any function you've assigned to its `onload` property, but only *after* the page is fully loaded. So, thank the designers of the window object for giving you a way to supply code that gets called after the page has loaded. Check this out:

The window object is built in to JavaScript. It represents the browser window.

```
<script>
function init() {
    let planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
}

window.onload = init;
</script>
```

First, create a function named init and put your existing code in the function.

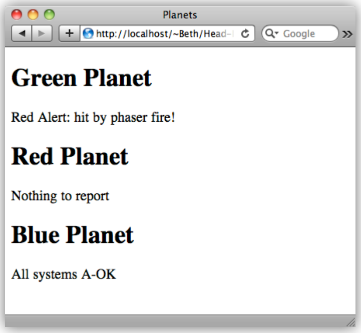
You can name this function anything you want, but it's often named init by convention.

Here's the code we had before, only now it's in the body of the init function.

Here, we're assigning the function init to the window.onload property. Make sure you don't use parentheses after the function name! We're not calling the function; we're just assigning the function value to the window.onload property.

Let's try that again...

Go ahead and reload the page with the new `init` function and the `onload` property. This time the browser will load the page completely, build the entire DOM, and *only then* call your `init` function.



Planets

http://localhost/~Beth/Head-

Green Planet

Red Alert: hit by phaser fire!

Red Planet

Nothing to report

Blue Planet

All systems A-OK

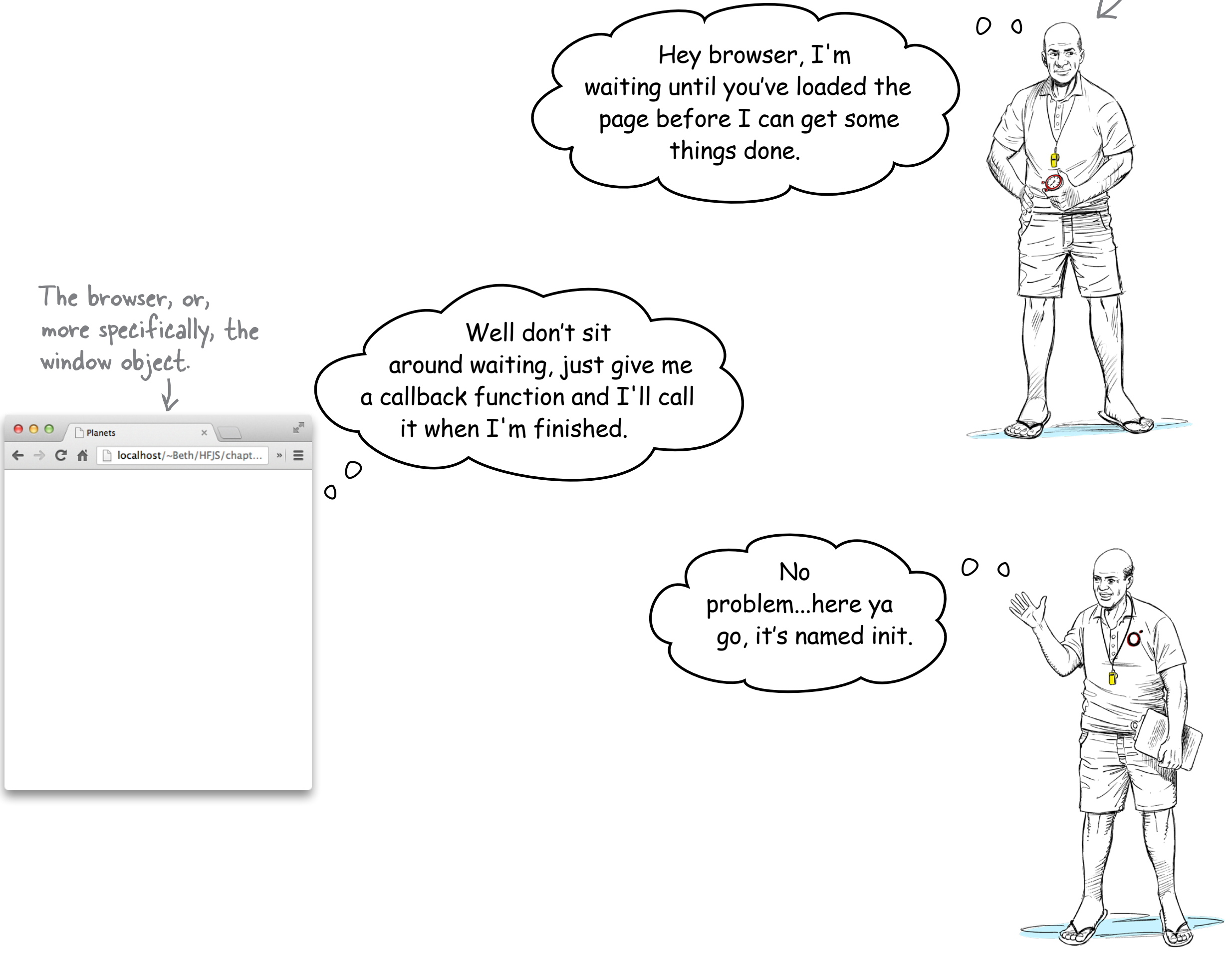
Ah, there we go, now the green planet shows the Red Alert, just like we wanted.

You say "event handler," I say "callback"

Let's think about how `onload` works just a bit more, because it uses a common coding pattern you'll see over and over again in JavaScript.

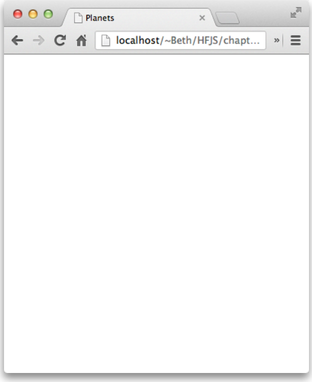
Suppose there's a big important event that's going to occur, and you *definitely* want to know about it. Say that event is the "page is loaded" event. Well, a common way to deal with that situation is through a *callback*, also known as an *event handler.*

A callback works like this: give a function to the object that knows about the event. When the event occurs, that object will call you back, or notify you, by calling that function. You're going to see this pattern in JavaScript for a variety of events.


Hey browser, I'm waiting until you've loaded the page before I can get some things done.
Planets
localhost/~Beth/HFJS/chapt...
Well don't sit around waiting, just give me a callback function and I'll call it when I'm finished.
No problem...here ya go, it's named init.

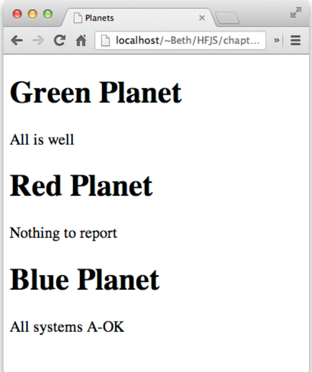
The callback, or event handler, if you prefer.

The browser, or, more specifically, the window object.


Planets
localhost/~Beth/HFJS/chapt...

Got it! I've set my onload property to init so I'll be sure to remember...

A bit later...


Planets
localhost/~Beth/HFJS/chapt...

Green Planet

All is well

Red Planet

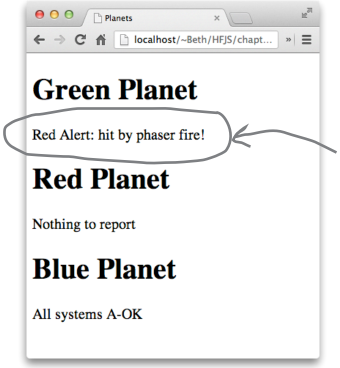
Nothing to report

Blue Planet

All systems A-OK

Whew, that was hard work, but the page is loaded. Let's see now...I need to call whatever my onload property is referencing... ah yes, the function init. Calling init!

...the init method is called and completes.


Planets
localhost/~Beth/HFJS/chapt...

Green Planet

Red Alert: hit by phaser fire!

Red Planet

Nothing to report

Blue Planet

All systems A-OK

And once init is called, we see the update to our page!

Hey browser, thanks for remembering to call init! Went off without a hitch!

Interesting. So I can use functions to package up code that can be called when some event happens. What other kinds of events can I handle with functions like this?

That's right, and there are many kinds of events you can handle if you want to. Some events, like the load event, are generated by the browser, while others are generated by a user interacting with the page, or even by your own JavaScript code.

You've seen an example of the "page is loaded" event, which we handle by setting the `onload` property of the window object. You can also write event handlers that handle things like "call this function every five seconds," or "some data arrived from a web service that we need to deal with," or "the user clicked a button and we need to get some data from a form," and there are many more. All these types of events are heavily used when you're creating pages that act more like applications than static documents (and who doesn't want that?). Right now, we've had just a brief glimpse of event handlers, but we'll be spending a lot more time on them a bit later given their important role in JavaScript programming.

Sharpen your pencil

Here's some HTML for a playlist of songs—but the list is empty. It's your job to complete the JavaScript below to add the songs to the list. Fill in the blanks with the JavaScript that will do the job. Check your answer at the end of the chapter before you go on.

Here's the HTML for the page.

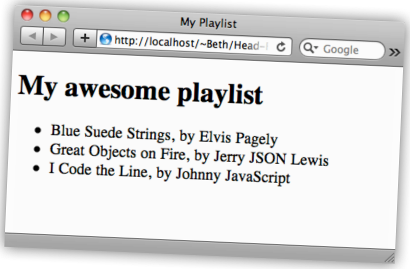
```
<!doctype html>
<html lang="en">
<head>
  <title>My Playlist</title>
  <meta charset="utf-8">
  <script>
  ____________ addSongs() {
    let song1 = document.______________("________");
    let _____ = _______________________("________");
    let _____ = ________.getElementById("________");

    ________.innerHTML = "Blue Suede Strings, by Elvis Pagely";
    __________________ = "Great Objects on Fire, by Jerry JSON Lewis";
    song3.____________ = "I Code the Line, by Johnny JavaScript";
  }
  window.___________ = ____________;
  </script>
</head>
<body>
  <h1>My awesome playlist</h1>
  <ul id="playlist">
    <li id="song1"></li>
    <li id="song2"></li>
    <li id="song3"></li>
  </ul>
</body>
</html>
```

Here's our script. This code should fill in the list of songs below, in the <ul>.

Fill in the blanks with the missing code to get the playlist filled out.

Here's the empty list of songs. The code above should add content to each <li> in the playlist.

When you get the JavaScript working, this is what you'll see when you load the page.



My awesome playlist

- Blue Suede Strings, by Elvis Pagely
- Great Objects on Fire, by Jerry JSON Lewis
- I Code the Line, by Johnny JavaScript

Solution on page 272

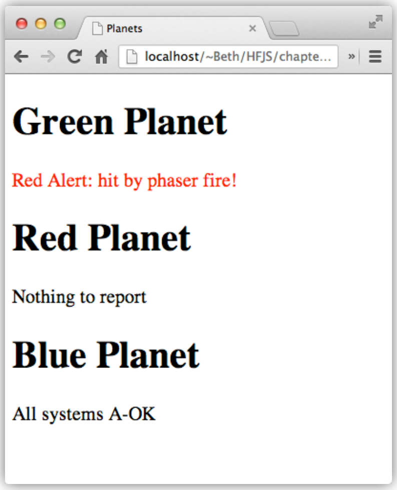
Why stop now? Let's take it further...

Let's think for a second about what you just did: you took a static web page and you dynamically changed the content of one of its paragraphs *using code*. It seems like a simple step, but this is really the beginning of making a *truly interactive* page.

And that's our goal, which we'll fully realize in Chapter 8.

Let's take the second step: now that you know how to get your hands on an element in the DOM, we'll set an *attribute* of an element with code.

Why would that be interesting? Well, take our simple planets example. When we set the greenplanet paragraph to read "Red Alert," we could also set the paragraph's color to red. That would certainly more clearly communicate our message.



Green Planet

Red Alert: hit by phaser fire!

Red Planet

Nothing to report

Blue Planet

All systems A-OK

Here's how we're going to do that:

① We'll define a CSS rule for the class "redtext" that specifies a red color for the text of the paragraph. That way, any element we add to this class will have red text.

② Next, we'll add code to take the greenplanet paragraph element and add the class "redtext".

That's it. All we need now is to learn how to set an attribute of an element, and then we can write the code.

Sharpen your pencil

How about getting another part of your brain working? We're going to need the CSS style for the class "redtext" that sets the color to "red" for the text in the greenplanet paragraph. If it's been a while since you wrote CSS, don't worry; give it a shot anyway. If you can do it in your sleep, awesome. Either way, you'll find the answer at the end of this chapter.

Solution on page 273

How to set an attribute with setAttribute

Element objects have a method named `setAttribute` that you can call to set the value of an HTML element's attribute. The `setAttribute` method looks like this:

We take our element object.

```
planet.setAttribute("class", "redtext");
```

Note that if the attribute doesn't exist, a new one will be created in the element.

And we use its setAttribute method to either add a new attribute or change an existing attribute.

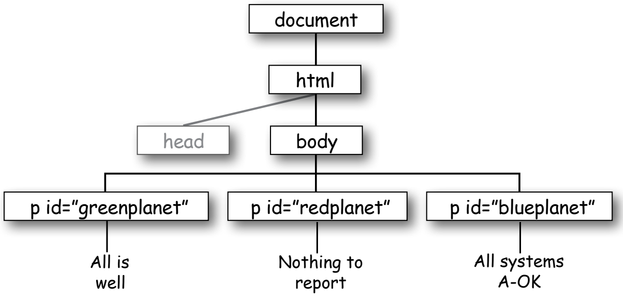
The method takes two arguments: the name of the attribute you want to set or change...

...and the value you'd like to set that attribute to.

We can call `setAttribute` on any element to change the value of an existing attribute or, if the attribute doesn't already exist, to add a new attribute to the element. As an example, let's check out how executing the code above affects our DOM:

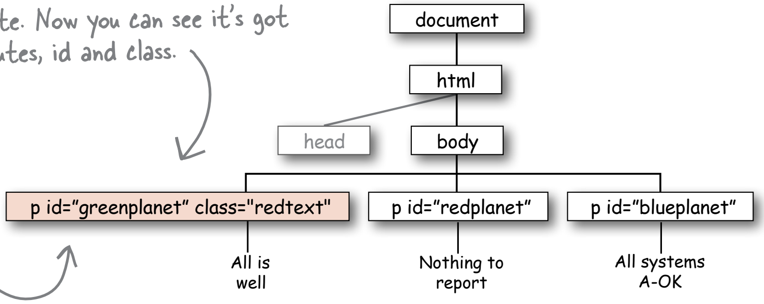
Before...

Here's the element before we call the setAttribute method on it. Notice this element already has one attribute, id.


document
html
head
body
p id="greenplanet"
p id="redplanet"
p id="blueplanet"
All is well
Nothing to report
All systems A-OK

...and After

And here's the element after we call setAttribute. Now you can see it's got two attributes, id and class.


document
html
head
body
p id="greenplanet" class="redtext"
p id="redplanet"
p id="blueplanet"
All is well
Nothing to report
All systems A-OK

Remember, when we call the setAttribute method, we're changing the element object in the DOM, which immediately changes what you see displayed in the browser.

More fun with attributes!

(You can GET attributes too)

Need to know the value of an attribute in an element? No problem, we have a `getAttribute` method that you can call to get the value of an HTML element's attribute:

```
let scoop = document.getElementById("raspberry");
let altText = scoop.getAttribute("alt");
console.log("I can't see the image in the console,");
console.log(" but I'm told it looks like: " + altText);
```

Get a reference to the element with getElementById, then use the element's getAttribute method to get the attribute.

Pass in the name of the attribute you want the value of.

What happens if my attribute doesn't exist in the element?

Remember what happens when you call `getElementById` and the id doesn't exist in the DOM? You get null. Same thing with `getAttribute`. If the attribute doesn't exist, you'll get back null. Here's how you test for that:

```
let scoop = document.getElementById("raspberry");
let altText = scoop.getAttribute("alt");
if (altText == null) {
    console.log("Oh, I guess there isn't an alt attribute.");
} else {
    console.log("I can't see the image in the console,");
    console.log(" but I'm told it looks like " + altText);
}
```

Test to make sure there actually is an attribute value returned.

If there's no attribute value, we do this...

...and if there is one, we can show the text content in the console.

Don't forget getElementById can return null too!

Any time you ask for something, you need to make sure you got back what you expected...

The call to getElementById can return a null value if the element id does not exist in the DOM. So, to follow best practices, you'll want to make sure you test for null after getting elements too. We could follow that rule ourselves, but then the book would end up being 1,000 pages longer.

Meanwhile, back at the ~~ranch~~ solar system...

It's time to put all the code for the planets together and do a final test drive.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <style>
    .redtext { color: red; }
  </style>
  <script>
    function init() {
        let planet = document.getElementById("greenplanet");
        planet.innerHTML = "Red Alert: hit by phaser fire!";
        planet.setAttribute("class", "redtext");
    }
    window.onload = init;
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

Here's all the HTML, CSS, and JavaScript for the planets.

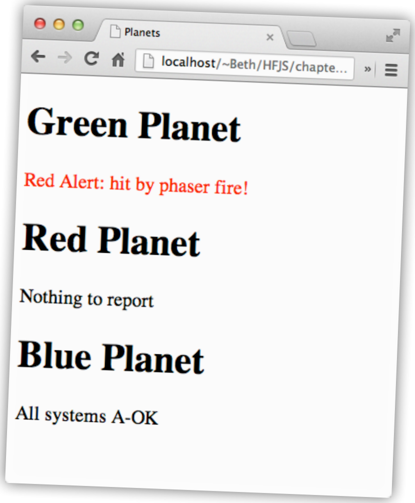
We've got the redtext class included here so when we add "redtext" as the value for the class attribute in our code, it turns the text red.

To review: we're getting the greenplanet element and stashing the value in the planet variable. Then we're changing the content of the element, and finally adding a class attribute that will turn the text of the element red.

We're calling the init function only when the page is fully loaded!

Test driving the planets one last time...

Load this page up in your browser, and you'll see the green planet has been hit by phaser fire...and now we see the message in bright red, so we'll be sure not to miss it!



Green Planet

Red Alert: hit by phaser fire!

Red Planet

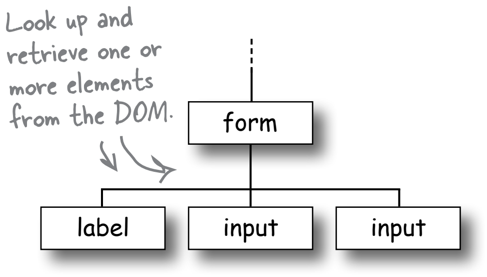
Nothing to report

Blue Planet

All systems A-OK

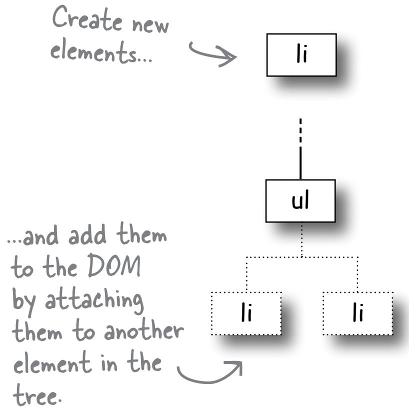
So what else is a DOM good for, anyway?

The DOM can do a fair bit more than we've seen so far, and we'll be seeing some of its other functionality as we move forward in the book. Let's just take a quick look for now, so you've got it in the back of your mind. You can:


Look up and retrieve one or more elements from the DOM.
form
label
input
input

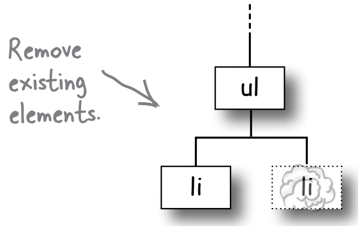
Get elements from the DOM.

Of course, you already know this because we've been using document.getElementById, but there are other ways to get elements as well; in fact, you can use tag names, class names, and attributes to retrieve not just one element, but a whole set of elements (say, all elements in the class "on_sale"). And you can get form values the user has typed in, like the text of an input element.


Create new elements...
li
ul
li
li
...and add them to the DOM by attaching them to another element in the tree.

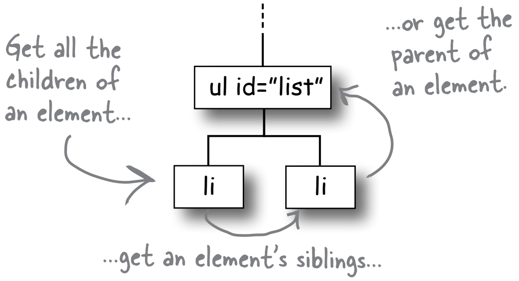
Create and add elements to the DOM.

You can create new elements, and you can also add those elements to the DOM. Of course, any changes you make to the DOM will show up immediately as the DOM is rendered by the browser (which is a good thing!).


Remove existing elements.
ul
li
li

Remove elements from the DOM.

You can also remove elements from the DOM by taking a parent element and removing any of its children. Again, you'll see the element removed in your browser window as soon as it is deleted from the DOM.

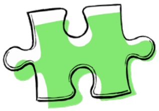

Get all the children of an element...
ul id="list"
li
li
...or get the parent of an element.
...get an element's siblings...

Traverse the elements in the DOM.

Once you have a handle to an element, you can find all its children, you can get its siblings (all the elements at the same level), and you can get its parent. The DOM is structured just like a family tree!

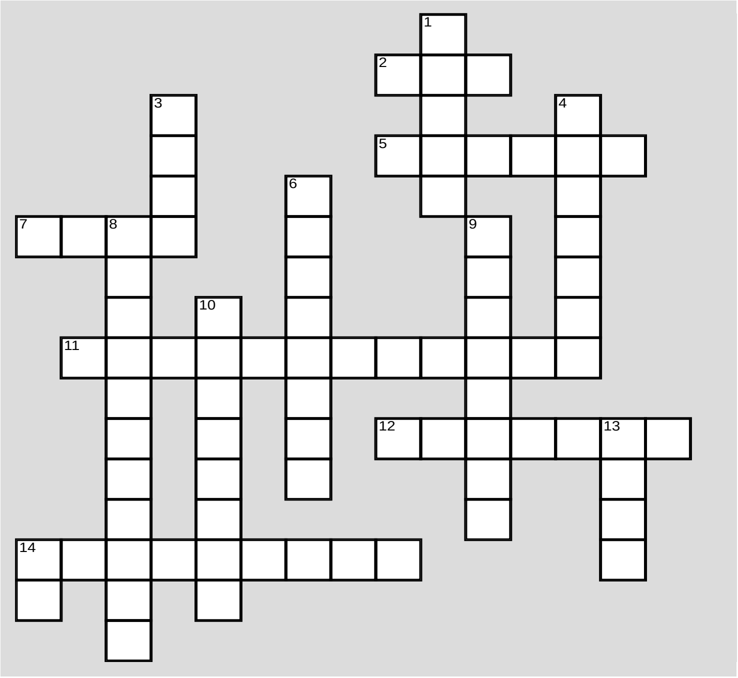
Bullet Points

- The **document object model**, or DOM, is the browser's representation of your web page.
- The browser creates the DOM for your page as it loads and parses the HTML.
- You get access to the DOM in your JavaScript code with the document object.
- The document object has properties and methods you can use to access and modify the DOM.
- The **document.getElementById** method grabs an element from the DOM using its id.
- The document.getElementById method returns an **element object** that represents an element in your page.
- An element object has properties and methods you can use to read an element's content, and change it.
- The **innerHTML** property holds the text content, as well as all nested HTML content, of an element.
- You can modify the content of an element by changing the value of its innerHTML property.
- When you modify an element by changing its innerHTML property, you see the change in your web page immediately.
- You can get the value of an element's attributes using the **getAttribute** method.
- You can set the value of an element's attributes using the **setAttribute** method.
- If you'd like to retrieve elements with CSS selectors, you can use the **querySelector** and **querySelectorAll** methods.
- If you put your code in a <script> element in the <head> of your page, you need to make sure you don't try to modify the DOM until the page is fully loaded.
- You can use the window object's **onload** property to set an **event handler**, or callback, function for the load event.
- The event handler for the window's onload property will be called as soon as the page is fully loaded.
- There are many different kinds of events we can handle in JavaScript with event handler functions.



JavaScript cross

Load the DOM into your brain with this puzzle.



ACROSS

2. When you load a page into the browser, the browser creates a _______ representing all the elements and content in the page.

5. getElementById is a _______ of the document object.

7. It's important to make sure the _______ is completely loaded before using code to get or change elements in the page.

11. Change the class of an element using the _______ method.

12. The setAttribute method is a method of an _______ object.

14. Use the element object's _______ property to change the HTML inside an element.

DOWN

1. Dr. Evel's passcode clue was in the element with the id _______.

3. The DOM is shaped like a _______.

4. Use the _______ to see if you have errors in your code.

6. Assign a _______ to the window.onload property to handle the load event.

8. Which planet gets hit by phaser fire?

9. The _______ object is always at the top of the DOM tree.

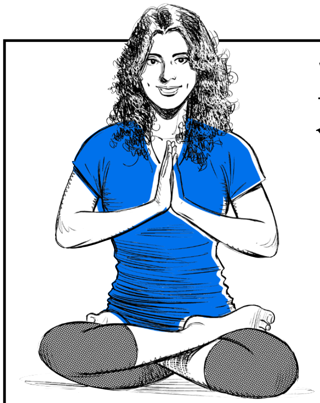
10. Functions that handle events are known as event _______.

13. It's a good idea to check for _______ when using getElementById.

14. The getElementById method gets an element by its _______.

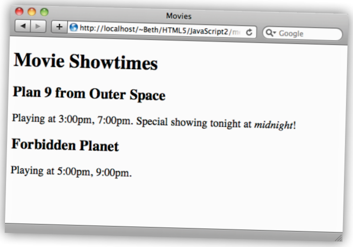
→ Solution on page 273

From page 246



BE the Browser Solution

Your job is to act like you're the browser. You need to parse the HTML and build your very own DOM from it. Go ahead and parse the following HTML, and draw your DOM below. Here's our solution.


Movies
http://localhost/~Beth/HTML5/JavaScript2/m
Google

Movie Showtimes

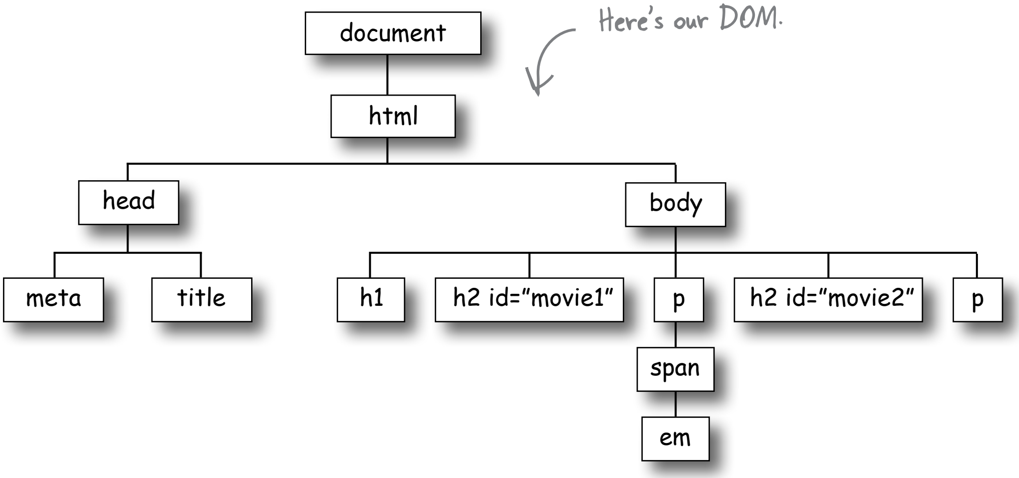
Plan 9 from Outer Space

Playing at 3:00pm, 7:00pm. Special showing tonight at *midnight*!

Forbidden Planet

Playing at 5:00pm, 9:00pm.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Movies</title>
  </head>
  <body>
    <h1>Movie Showtimes</h1>
    <h2 id="movie1" >Plan 9 from Outer Space</h2>
    <p>Playing at 3:00pm, 7:00pm.
      <span>
        Special showing tonight at <em>midnight</em>!
      </span>
    </p>
    <h2 id="movie2">Forbidden Planet</h2>
    <p>Playing at 5:00pm, 9:00pm.</p>
  </body>
</html>
```


document
html
head
body
meta
title
h1
h2 id="movie1"
p
h2 id="movie2"
p
span
em

Here's our DOM.

Sharpen your pencil Solution

From page 263

Here's some HTML for a playlist of songs—but the list is empty. It's your job to complete the JavaScript below to add the songs to the list. Fill in the blanks with the JavaScript that will do the job. Here's our solution.

Here's the HTML for the page.

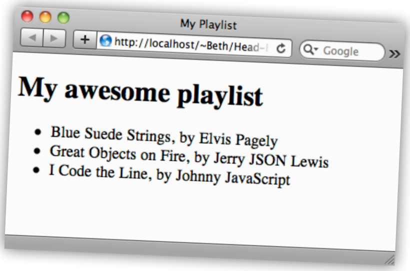
Here's our script. This code should fill in the list of songs below, in the <ul>.

Fill in the blanks with the missing code to get the playlist filled out.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My Playlist</title>
  <script>
  function addSongs() {
    let song1 = document.getElementById("song1");
    let song2 = document.getElementById("song2");
    let song3 = document.getElementById("song3");

    song1.innerHTML = "Blue Suede Strings, by Elvis Pagely";
    song2.innerHTML = "Great Objects on Fire, by Jerry JSON Lewis";
    song3.innerHTML = "I Code the Line, by Johnny JavaScript";
  }
  window.onload = addSongs;
  </script>
</head>
<body>
  <h1>My awesome playlist</h1>
  <ul id="playlist">
    <li id="song1"></li>
    <li id="song2"></li>
    <li id="song3"></li>
  </ul>
</body>
</html>
```

Here's the empty list of songs. The code above should add content to each <li> in the playlist.

When you get the JavaScript working, this is what you'll see when you load the page.



My Playlist

http://localhost/~Beth/Head-

Google

My awesome playlist

- Blue Suede Strings, by Elvis Pagely
- Great Objects on Fire, by Jerry JSON Lewis
- I Code the Line, by Johnny JavaScript

Sharpen your pencil Solution

From page 264

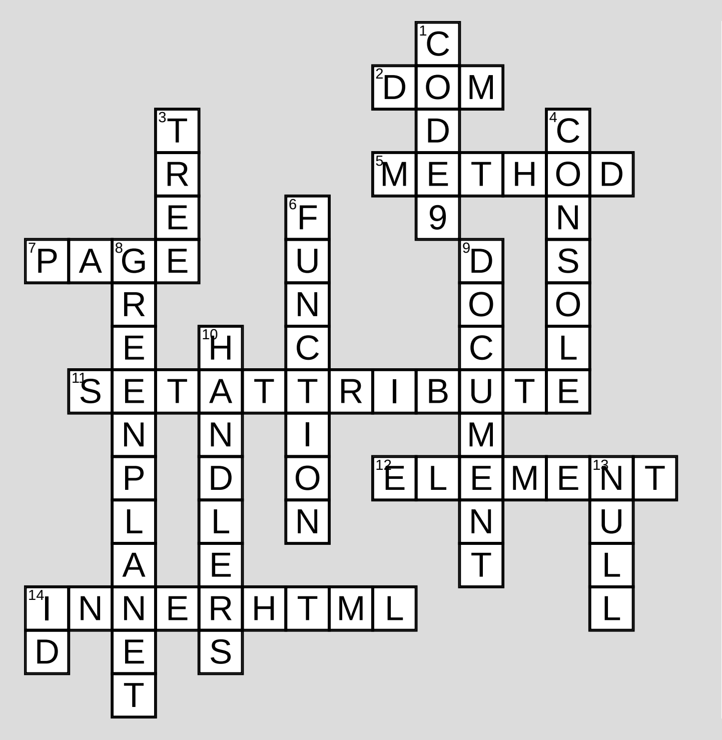
How about getting another part of your brain working? We're going to need the CSS style for the class "redtext" that sets the color to "red" for the text in the greenplanet paragraph. If it's been a while since you wrote CSS, don't worry; give it a shot anyway. If you can do it in your sleep, awesome. Here's our solution.

```
.redtext { color: red; }
```

JavaScript cross Solution

From page 270

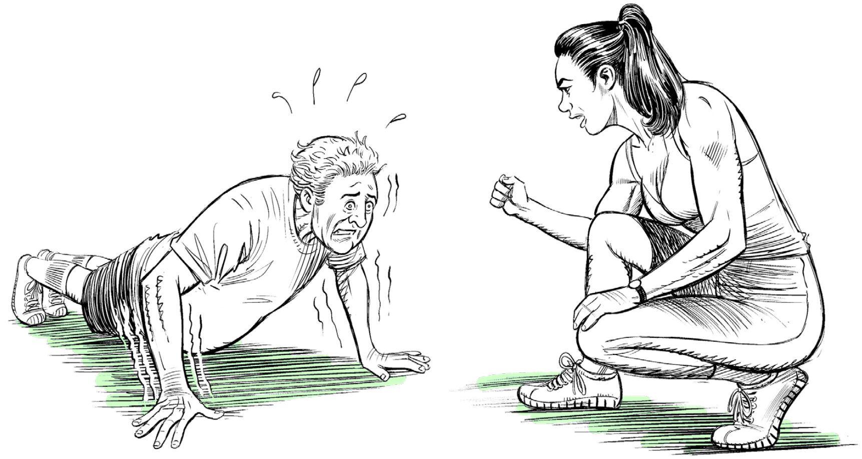
Load the DOM into your brain with this puzzle. Here's our solution.



									1 C					
								2 D	O	M				
			3 T						D			4 C		
			R					5 M	E	T	H	O	D	
			E			6 F			9			N		
7 P	A	8 G	E			U				9 D		S		
		R				N				O		O		
		E		10 H		C				C		L		
	11 S	E	T	A	T	T	R	I	B	U	T	E		
		N		N		I				M				
		P		D		O		12 E	L	E	M	E	13 N	T
		L		L		N				N			U	
		A		E						T			L	
14 I	N	N	E	R	H	T	M	L					L	
D		E		S										
		T												

7 types, equality, conversion, and all that jazz

Serious types

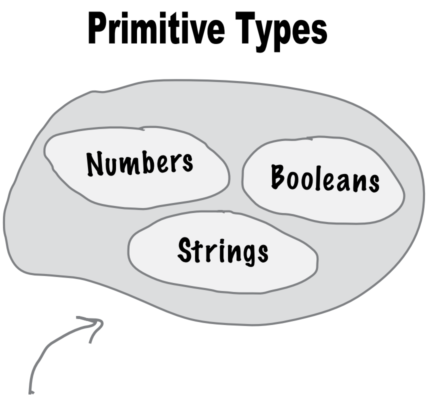


It's time to get serious about our types. One of the great things about JavaScript is you can get a long way without knowing a lot of details of the language. But to truly **master the language**, get that promotion, and get on to the things you really want to do in life, you have to rock at **types**. Remember what we said about JavaScript back in Chapter 1? That it didn't have the luxury of a silver-spoon, academic, peer-reviewed language definition? Well, that's true, but the lack of an academic life didn't stop Steve Jobs and Bill Gates, and it didn't stop JavaScript either. It does mean that JavaScript doesn't have the...well, the most thought-out type system, and we'll find a few **idiosyncrasies** along the way. But don't worry, in this chapter we're going to nail all that down, and soon you'll be able to avoid all those embarrassing moments with types.

The truth is out there...

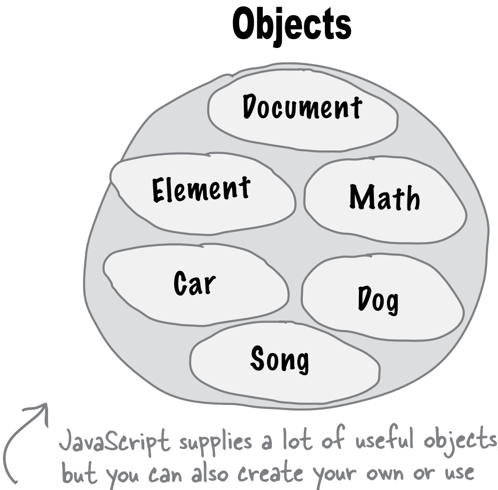
Now that you've had some experience working with JavaScript types—there's your primitives with numbers, strings, and booleans, and there's all the objects, some supplied by JavaScript (like the `Math` object), some supplied by the browser (like the `document` object), and some you've written yourself—aren't you just basking in the glow of JavaScript's simple, powerful, and consistent type system?

Low-level basic types for numbers, strings, and booleans.



These are all supplied by JavaScript. There are a few more that we'll get to a bit later in the book.

High-level objects used to represent the things in your problem space.



JavaScript supplies a lot of useful objects, but you can also create your own or use objects other developers have written.

After all, what else would you expect from the official language of Webville? In fact, if you were a mere scripter, you might think about sitting back, sipping on that Webville Martini, and taking a much-needed break…

But you're not a mere scripter, and something is amiss. You have a sinking feeling that behind Webville's picket fences something bizarre is at work. You've heard the reports of sightings of strings that are acting like objects, you've read in the blogs about a (probably radioactive) null type, you've heard the rumors that the JavaScript interpreter as of late has been doing some weird type conversions... What does it all mean? We don't know, but the truth is out there, and we're going to uncover it in this chapter—and when we do, we might just turn what you think of as true and false upside down.

Who am I?

A bunch of JavaScript values and party crashers, in full costume, are playing a party game: "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Draw an arrow from each sentence to the name of one attendee. We've already guessed one of them for you. Check your answers at the end of the chapter before you go on.

If you find this exercise difficult, it's okay to cheat and look at the answers.

I get returned from a function when there is no return statement.

I'm the value of a variable when it hasn't been assigned a value.

I'm the value of an array item that doesn't exist in a sparse array.

I'm the value of a property that doesn't exist.

I'm the value of a property that's been deleted.

Tonight's attendees:

zero

empty object

null

undefined

NaN

infinity

area 51

...---...

{}

[]

Solution on page 321

Watch out, you might bump into undefined when you aren't expecting it...

As you can see, whenever things get shaky—you need a variable that's not been initialized yet, you want a property that doesn't exist (or has been deleted), you go after an array item that isn't there—you're going to encounter `undefined`.

But what the heck is it? It's not really that complicated. Think of `undefined` as the value assigned to things that don't yet have a value (in other words, they haven't been initialized).

So what good is it? Well, `undefined` gives you a way to test to see if a variable (or property, or array item) has been given a value. Let's look at a couple of examples, starting with an unassigned variable:

```
let x;


if (x == undefined) {
    // x isn't defined! just deal with it!
}
```

You can check to see if a variable like x is undefined. Just compare it to the value undefined.

Note that we're using the value undefined here, not to be confused with the string "undefined".

Or how about an object property?

```
let customer = {
    name: "Jenny"
};
if (customer.phoneNumber == undefined) {
    // set the customer's phone number
}
```

You can check to see if a property is undefined, again by comparing it to the value undefined.

there are no Dumb Questions

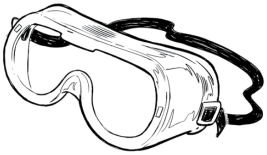
Q: When do I need to check if a variable (or property or array item) is undefined?

A: Your code design will dictate this. If you've written code so that a property or variable may not have a value when a certain block of code is executed, then checking for undefined gives you a way to handle that situation rather than computing with undefined values.

Q: If undefined is a value, does it have a type?

A: Yes, it does. The type of undefined is undefined. Why? Well, our logic (work with us here) is this: it isn't an object, or a number or a string or a boolean, or really anything that is defined. So why not make the type undefined, too? This is one of those weird twilight zones of JavaScript you just have to accept.

IN THE LABORATORY



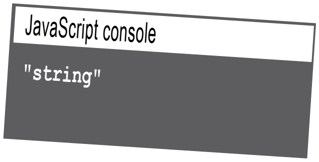
In the laboratory, we like to take things apart, look under the hood, poke and prod, hook up our diagnostic tools, and check out what is really going on. Today, we're investigating JavaScript's type system, and we've found a little diagnostic tool called **typeof** to examine variables. Put your lab coat and safety goggles on, and come on in and join us.

The **typeof** operator is built into JavaScript. You can use it to probe the type of its operand (the thing you use it to operate on). Here's an example:

```
let subject = "Just a string";

let probe = typeof subject;
console.log(probe);
```

The typeof operator takes an operand and evaluates to the type of the operand.

The type here is "string". Note that typeof uses strings to represent types, like "string", "boolean", "number", "object", "undefined", and so on.



JavaScript console

```
"string"
```

Now it's your turn. Collect the data for the following experiments:

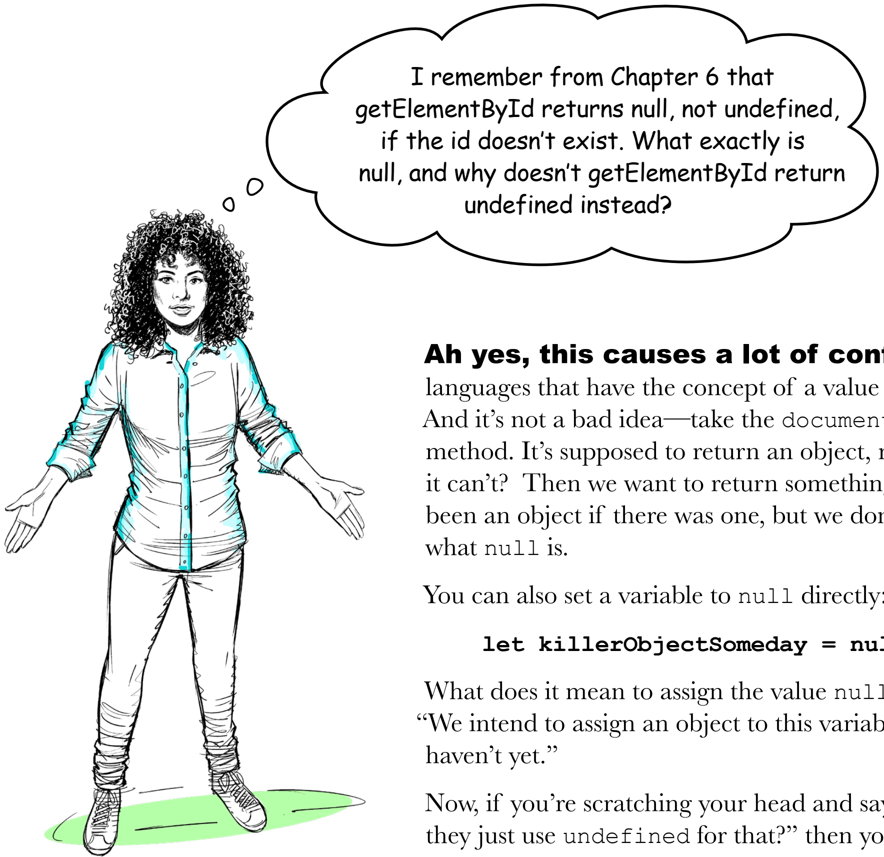
```
let test1 = "abcdef";
let test2 = 123;
let test3 = true;
let test4 = {};
let test5 = [];
let test6;
let test7 = {"abcdef": 123};
let test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Here's the test data, and the tests.

JavaScript console

Put your results here. Are there any surprises?

Solution on page 322


I remember from Chapter 6 that getElementById returns null, not undefined, if the id doesn't exist. What exactly is null, and why doesn't getElementById return undefined instead?

Ah yes, this causes a lot of confusion. There are many languages that have the concept of a value that means "no object." And it's not a bad idea—take the `document.getElementById` method. It's supposed to return an object, right? So, what happens if it can't? Then we want to return something that says "I would have been an object if there was one, but we don't have one." And that's what `null` is.

You can also set a variable to `null` directly:

```
let killerObjectSomeday = null;
```

What does it mean to assign the value `null` to a variable? How about "We intend to assign an object to this variable at some point, but we haven't yet."

Now, if you're scratching your head and saying "Hmm, why didn't they just use `undefined` for that?" then you're in good company. The answer comes from the very beginnings of JavaScript. The idea was to have one value for variables that haven't been initialized to anything yet, and another that means the lack of an object. It isn't pretty, and it's a little redundant, but it is what it is at this point. Just remember the intent of each (`undefined` and `null`), and know that it is most common to use `null` in places where an object should be but one can't be created or found, and it is most common to find `undefined` when you have a variable that hasn't been initialized, or an object with a missing property, or an array with a missing value.

BACK IN THE LABORATORY

Oops, we forgot null in our test data. Here's the missing test case:

```
let test10 = null;

console.log(typeof test10);
```

Put your results here.

JavaScript console

Solution on page 323

How to use null

There are many functions and methods out there in the world that return objects, and you'll often want to make sure what you're getting back is a full-fledged object, and not `null`, just in case the function wasn't able to find one or make one to return to you. You've already seen examples from the DOM where a test is needed:

Let's look for the all-important header element.

```
let header = document.getElementById("header");
if (header == null) {
     // okay, something is seriously wrong if we have no header
}
```

Uh-oh, it doesn't exist. Abandon ship!

Keep in mind that getting `null` doesn't necessarily mean something is wrong. It may just mean something doesn't exist yet and needs to be created, or something doesn't exist and you can skip it. Let's say users have the ability to open or close a weather widget on your site. If a user has it open there's a `<div>` with the id of "weatherDiv", and if not, there isn't. All of a sudden `null` becomes quite useful:

Let's see if the element with id "weatherDiv" exists.

```
let weather = document.getElementById("weatherDiv");
if (weather != null) {
     // create content for the weather div
}
```

If the result of getElementById isn't null, then there is such an element in the page. Let's create a nice weather widget for it (presumably getting the weather for the local area).

We can use null to check to see if an object exists yet or not.

Blaine, Missouri

It is always 67 degrees with a 40% chance of rain.

Remember, null is intended to represent an object that isn't there.

WickedlySmart's

Believe It or Not!!

The Number that isn't a Number

It's easy to write JavaScript statements that result in numeric values that are not well defined.

Here are a few examples:

```
let a = 0/0;
```

In mathematics this has no direct answer, so we can't expect JavaScript to know the answer either!

```
let b = "food" * 1000;
```

We don't know what this evaluates to, but it is certainly not a number!

```
let c = Math.sqrt(-9);
```

If you remember high school math, the square root of a negative number is an imaginary number, which you can't represent in JavaScript.

Believe it or not, there are numeric values that are **impossible to represent** in JavaScript! JavaScript can't express these values, so it has a stand-in value that it uses:

NaN

JavaScript uses the value NaN, more commonly known as "Not a Number," to represent numeric results that, well, can't be represented. Take 0 / 0 for instance. 0 / 0 evaluates to something that just can't be represented in a computer, so it is represented by NaN in JavaScript.

NaN MAY BE THE WEIRDEST VALUE IN THE WORLD. Not only does it represent all the numeric values that can't be represented, it is the only value in JavaScript that isn't equal to itself!

You heard that right. If you compare NaN to NaN, they aren't equal!

NaN != NaN

Dealing with NaN

Now, you might think that dealing with `NaN` is a rare event, but if you're working with any kind of code that uses numbers, you'll be surprised how often it shows up. The most common thing you'll need to do is test for `NaN`, and given everything you've learned about JavaScript, how to do this might seem obvious:

```
if (myNum == NaN) {
    myNum = 0;
}
```

You'd think this would work, but it doesn't.

WRONG!

Any sensible person would assume that's how you test to see if a variable holds a `NaN` value, but it doesn't work. Why? Well, `NaN` isn't equal to anything—not even itself—so, any kind of test for equality with `NaN` is off the table. Instead, you need to use a special function, `isNaN`. Like this:

```
if (isNaN(myNum)) {
    myNum = 0;
}
```

Use the isNaN function, which returns true if the value passed to it is not a number.

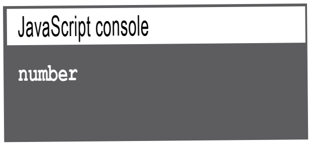
RIGHT!

It gets even weirder...

Let's think through this a bit more. If `NaN` stands for "Not a Number," what is it? Wouldn't it be easier if it were named for what it is rather than what it isn't? What do you think it is? We can check its type for a hint:

```
let test11 = 0 / 0;
console.log(typeof test11);
```

Here's what we got.



```
number
```

If your mind isn't blown, you should probably just use this book for some good kindling.

What on earth? `NaN` is of type number? How can something that's not a number have the type number? Okay, deep breath. Think of `NaN` as just a poorly named value. Someone should have called it something more like "number that can't be represented" (okay, we agree the acronym isn't quite as nice) instead of "Not a Number." If you think about it like that, then you can think of `NaN` as being a value that is a number but can't be represented (at least, not by a computer).

Go ahead and add this one to your JavaScript twilight zone list.

there are no Dumb Questions

Q: If I pass isNaN a string, which isn't a number, will it return true?

A: It sure will, just as you'd expect. You can expect a variable holding the value NaN, or any other value that isn't an actual number, to result in isNaN returning true (and false otherwise). There are a few caveats to this that you'll see when we talk about type conversion.

Q: But why isn't NaN equal to itself?

A: If you're deeply interested in this topic, you'll want to seek out the IEEE floating-point specification. However, the layman's insight into this is that NaN represents an unrepresentable numeric value, but not all unrepresentable numbers are equal. For instance, take sqrt(–1) and sqrt(–2). They are definitely not the same, but they both produce NaN.

Q: When we divide 0 / 0 we get NaN, but I tried dividing 10 / 0 and got Infinity. Is that different from NaN?

A: Good find. The Infinity (or –Infinity) value in JavaScript represents all numbers (to get a little technical) that exceed the upper limit on computer floating-point numbers, which is 1.7976931348623157e+308 (or –1.7976931348623157e+308 for –Infinity). The type of Infinity is number, and you can test for it if you suspect one of your values is getting a little large:

```
     if (tamale == Infinity) {
       alert("That's a big tamale!");
   }
```

Q: You did blow my mind with that "NaN is a number" thing. Any other mind-blowing details?

A: Funny you should ask. How about Infinity minus Infinity equals...wait for it...NaN. We'll refer you to a good mathematician to understand that one.

Q: Just to cover every detail, did you say what the type of null is?

A: No, but a quick way to find out is by using the typeof operator on null. If you do that, you'll get back the result "object". This makes sense, from the perspective that null is used to represent an object that isn't there. However, this point has been heavily debated, and the most recent spec defines the type of null as null. You'll find this an area where your browser's JavaScript implementation may not match the spec, but, in practice you'll rarely need to use the type of null in code.

Exercise

We've been looking at some rather, um, interesting values so far in this chapter. Now, let's take a look at some interesting behavior. Try adding the code below to the <script> element in a basic web page and see what you get in the console when you load up the page. You probably won't get why yet, but see if you can take a guess about what might be going on.

```
if (99 == "99") {
    console.log("A number equals a string!");
} else {
    console.log("No way a number equals a string");
}
```

JavaScript console

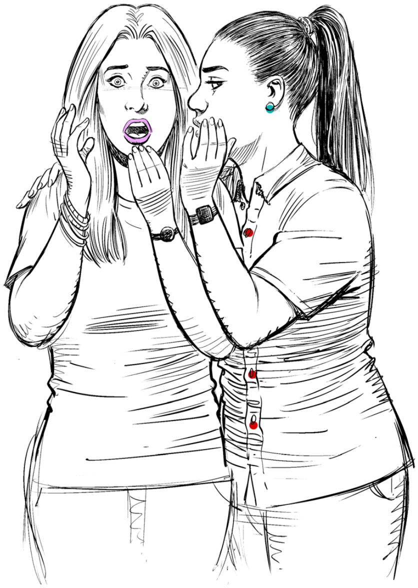
Write what you get here.

→ Solution on page 323

We have a confession to make

There is an aspect of JavaScript we've deliberately been holding back on. We could have told you up front, but it wouldn't have made as much sense as it will now.

It's not so much that we've been pulling the wool over your eyes, it's that there is more to the story than we've been telling you. And what is this topic? Here, let's take a look:



At some point a variable gets set, in this case to the number 99.

```
let testMe = 99;
```

And later it gets compared with a number in a conditional test.

```
if (testMe == 99) {
    // good things happen
}
```

Straightforward enough? Sure, what could be easier? However, one thing we've done at least once so far in this book, which you might not have noticed, is something like this:

At some point a variable gets set, in this case to the string "99".

Did we mention we're using a string this time?

```
let testMe = "99";
```

And later it gets compared with a number in a conditional test.

```
if (testMe == 99) {
    // good things happen
}
```

Now we have a string being compared to a number.

So what happens when we compare a number to a string? Mass chaos? Computer meltdown? Rioting in the streets?

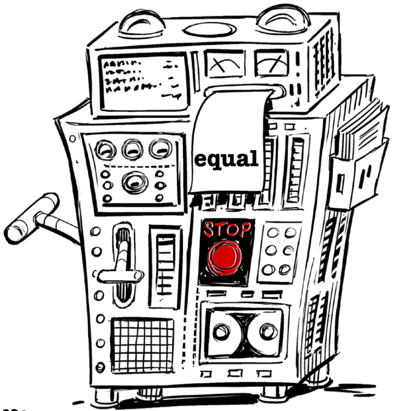
No, JavaScript is smart enough to determine that 99 and "99" are the same for all practical purposes. But what exactly is going on behind the scenes to make this work? Let's take a look…

Bullet Points

Just a quick reminder about the difference between assignment and equality:

- `let x = 99;`
  = is the assignment operator. It is used to assign a value to a variable.
- `x == 99`
  == is a comparison operator. It is used to compare one value with another to see if they're equal.

Understanding the equality operator
(otherwise known as ==)



You'd think that understanding equality would be a simple topic. After all, 1 == 1, "guacamole" == "guacamole," and true == true. But clearly there is more at work here if "99" == 99. What could be going on inside the equality operator to make that happen?

It turns out the == operator takes the types of its operands (that is, the two things you're comparing) into account when it does a comparison. You can break this down into two cases:

If the two values have the same type, just compare them

If the two values you are comparing have the same type, like two numbers or two strings, then the comparison works just like you would expect: the two values are compared against each other and the result is true if they are the same value. Easy enough.

If the two values have different types, try to convert them into the same type and then compare them

This is the more interesting case. Say you have two values with different types that you want to compare, like a number and a string. What JavaScript does is convert the string into a number and then compare the two values, like this:

Note that the conversion is only temporary, so that the comparison can happen.

```
99 == "99"
```

When you're comparing a number and a string, JavaScript converts the string to a number (if possible)...

```
99 == 99
```

...and then tries the comparison again. If they're equal the expression results in true, and otherwise false.

Okay, that makes some intuitive sense, but what are the rules here? What if we compare a boolean to a number, or null to undefined, or some other combination of values? How do we know what's going to get converted into what? And why not convert the number into a string instead, or use some other scheme to test their equality? Well, this is defined by a fairly simple set of rules in the JavaScript specification that determine how the conversion happens when we compare two values with different types. This is one of those things you just need to internalize—once you've done that, you'll be on top of how comparisons work for the rest of your JavaScript career.

This will also set you above your peers, and help you nail your next interview.

How equality converts its operands
(sounds more dangerous than it actually is)

You now know that when you compare two values that have different types, JavaScript will convert one type into another in order to compare them. If you're coming from another language this might seem strange, given that this is typically something you'd have to code explicitly rather than having it happen automatically. But no worries; in general, it's a useful thing in JavaScript *so long as you understand when and how it happens*. And that's what we've got to figure out now—when it happens and how it happens.

Here we go (in four simple cases):

Case #1: Comparing a number and a string

If you're comparing a string and a number, the same thing happens every time: the string is converted into a number, and the two numbers are then compared. This doesn't always go well, because not all strings can be converted to numbers. Let's see what happens in that case:

```
99 == "vanilla"
 99 == NaN
   false
```

Once again, we're comparing a number and a string. But this time, when we try to convert the string to a number, we fail.

When we try to convert "vanilla" to a number, we get NaN, and NaN isn't equal to anything. So, the result is false.

Case #2: Comparing a boolean with any other type

In this case, we convert the boolean to a number and compare. This might seem a little strange, but it's easier to digest if you just remember that true converts to 1 and false converts to 0. You also need to understand that sometimes this case requires doing more than one type conversion. Let's look at a few examples:

```
1 == true
1 == 1
 true
```

We're comparing a number and a boolean. The true value is converted to the number 1.

And then we compare 1 to 1, which is true.

Here's another case, this time comparing a boolean to a string.
Notice how more steps are needed:

```
"1" == true
```

We're comparing a string and a boolean. The true value is first converted to the number 1.

```
"1" == 1
```

And then we compare the string "1" to 1.

```
1 == 1
```

Now we use the rule from case #1, and the string is converted into a number.

```
true
```

And now we can finally compare a number to a number; in this case, the result is true.

Case #3: Comparing null and undefined

Comparing these values evaluates to true. That might seem odd as well, but it's the rule. For some insight, these values both essentially represent "no value" (that is, a variable with no value, or an object with no value), so they are considered to be equal:

```
undefined == null
```

```
true
```

The values undefined and null are always equal.

Case #4: Oh, actually there is no case #4

That's it. You can pretty much determine the value of any equality with the previous rules. That said, there are a few edge cases and caveats. One caveat is that we still need to talk about comparing objects, which we'll get to in a bit. The other is around conversions that might catch you off guard. Here's one example:

```
1 == ""
```

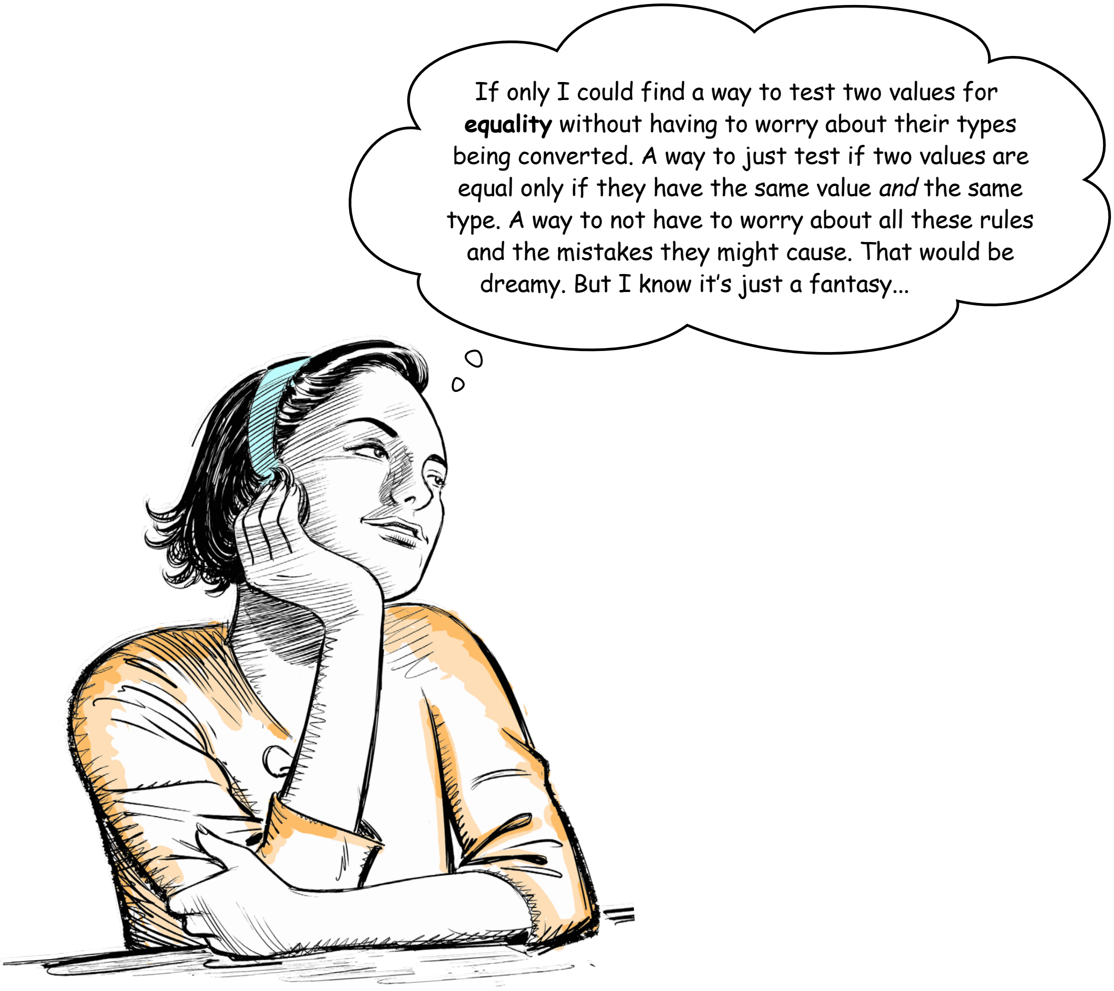
We're comparing a number and a string (case #1).

```
1 == 0
```

The empty string is converted to the number 0...Believe it or not!

```
false
```

Ah, too bad, 1 and 0 are not the same. So this evaluates to false.


If only I could find a way to test two values for **equality** without having to worry about their types being converted. A way to just test if two values are equal only if they have the same value *and* the same type. A way to not have to worry about all these rules and the mistakes they might cause. That would be dreamy. But I know it's just a fantasy...

How to get strict with equality

While we're making confessions, here's another one: there is not one but *two equality operators*. You've already been introduced to == (equality), and the other operator is === (strict equality).

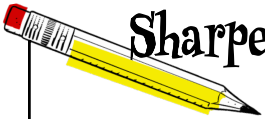
That's right, three equals. You can use === in place of == anytime you want, but before you start doing that, let's make sure you understand how they differ.

With ==, you now know all the complex rules around how the operands are converted (if they're different types) when they're compared. With ===, the rules are even more complicated.

Just kidding; actually, there is *only one rule* with ===:

Two values are strictly equal only if they have the same type and the same value.

Read that again. What that means is that if two values have the same type, we compare them. If they don't, forget it, we're calling it false no matter what—no conversion, no figuring out complex rules, none of that. All you need to remember is that === will find two values equal *only if they are the same type and the same value.*



Sharpen your pencil

Write true or false below the operators == and === to represent the result of each of the following comparisons.

	==	===	
`"42" == 42`	true		`"42" === 42`
`"0" == 0`			`"0" === 0`
`"0" == false`			`"0" === false`
`"true" == true`			`"true" === true`
`true == (1 == "1")` Tricky!			`true === (1 === "1")` Tricky!

Solution on page 324

there are no Dumb Questions

Q: What happens if I compare a number, like 99, to a string, like "ninety-nine", that can't be converted to a number?

A: JavaScript will try to convert "ninety-nine" to a number, and it will fail, resulting in NaN. So the two values won't be equal, and the result will be false.

Q: How does JavaScript convert strings to numbers?

A: It uses an algorithm to parse the individual characters of a string and try to turn each one of them into a number. So if you write "34", it will look at "3," and see that can be a 3, and then it will look at "4" and see that can be a 4. You can also convert strings like "1.2" to floating-point numbers—JavaScript is smart enough to recognize a string like this can still be a number.

Q: So, what if I try something like "true" == true?

A: That is comparing a string and a boolean, so according to the rules, JavaScript will first convert true to 1, and then compare "true" and 1. It will then try to convert "true" to a number and fail, so you'll get false.

Q: So if there is both an == and a === operator, does that mean we have <= and <==, and >= and >==?

A: No. There are no <== and >== operators. You can use only <= and >=. These operators only know how to compare strings and numbers (true <= false doesn't really make sense), so if you try to compare any values other than two strings or two numbers (or a string and a number), JavaScript will attempt to convert the types using the rules we've discussed.

Q: If I write 99 <= "100", what happens?

A: Use the rules: "100" is converted to a number and then compared with 99. Because 99 is less than or equal to 100 (it's less than), the result is true.

Q: Is there a !== operator?

A: Yes, and just like === is stricter than ==, !== is stricter than !=. You use the same rules for !== as you do for ===, except that you're checking for inequality instead of equality.

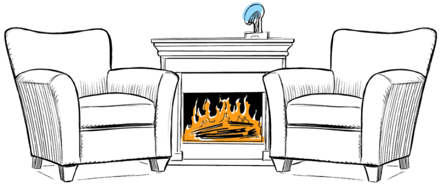
Q: Do we use the same rules when we're comparing, say, a boolean and a number with < and >, like 0 < true?

A: Yup! And in that case, true gets converted to 1, so you'll get true because 0 is less than 1.

Q: It makes sense for a string to be equal to another string, but how can a string be less than or greater than another string?

A: Good question. What does it mean to say "banana" < "mango"? Well, with strings, you can use alphabetical order to tell if one string is less than or greater than another. Since "banana" begins with a "b" and "mango" with an "m", "banana" is less than "mango" because "b" comes before "m" in the alphabet. And "mango" is less than "melon" because, while the first letters are the same, when we compare the second letters, "a" comes before "e".

This alphabetical comparison can trip you up, however; for instance, "Mango" < "mango" is true, even though you might think that "M" is greater than "m" because its "M" is capitalized. The ordering of strings has to do with the ordering of the Unicode values that are used to represent each character in the computer (Unicode is a standard for representing characters digitally), and that ordering might not always be what you expect! For all the details, try googling "Unicode." But most of the time, the basic alphabetical ordering is all you need to know if one string is less than or greater than another.

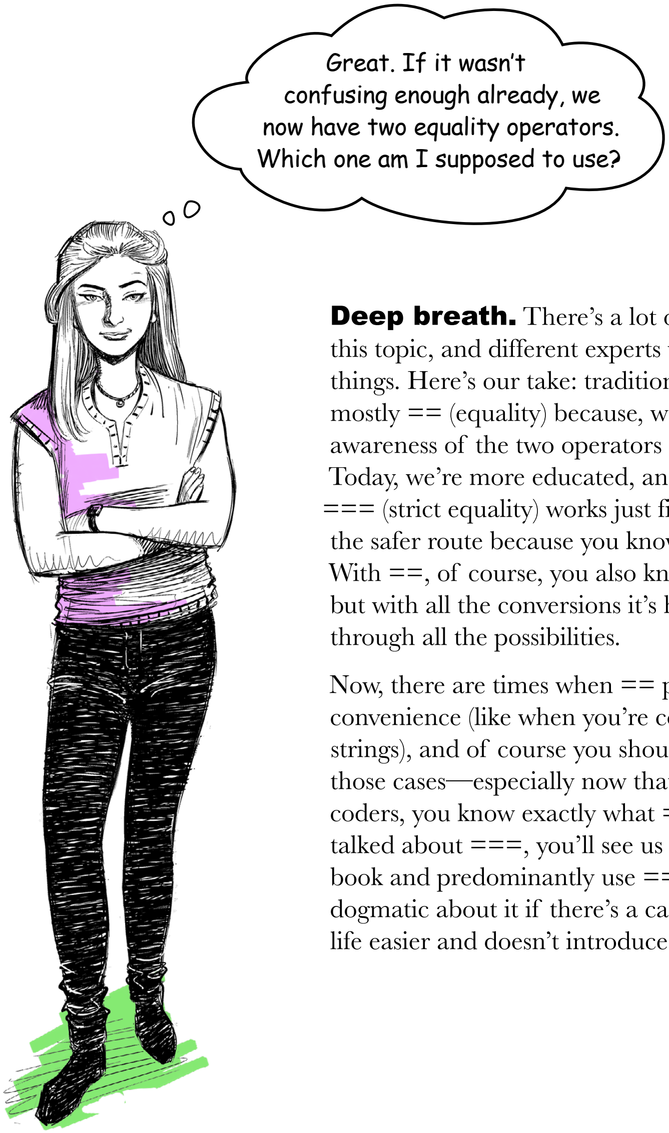


Fireside Chats

Tonight's talk: **The equality and strict equality operators let us know who is boss.**

==	===
Ah, look who it is, Mr. Uptight.	Just keep in mind that several leading JavaScript gurus say that developers should use me, and only me. They think you should be taken out of the language altogether.
I'm up for a count of == versus === across all JavaScript code out in the world. You're going to come in way behind. It won't even be close.	You know, you might be right, but folks are slowly starting to get it, and those numbers are changing.
I don't think so. I provide a valuable service. Who doesn't want to, say, compare user input in the form of a string to a number every once in a while?	And with it come all the rules you have to keep in mind to even use ==. Keep life and code simple; use ===, and if you need to convert user input to a number there are methods for that.
When you were in grade school did you have to walk to school in the snow, every day, uphill, in both directions? Do you always have to do things the hard way?	Very funny. There's nothing wrong with being strict and having clear-cut semantics around your comparisons. Bad, unexpected things can happen if you don't keep all the rules in mind.
The thing is, not only can I do the same comparisons you do, but I add value on top of that by doing some nice conversions of types.	Every time I look at your rules I throw up in my mouth a little. I mean, comparing a boolean to anything means I convert the boolean to a number? That doesn't seem very sensical to me.
You'd rather just throw your hands up, call it false, and go home?	

==	===
	No, but one can get a little too lax around your complex rules.
It's working so far. Look at all the code out there, a lot written by mere…well, *scripters*.	
	That's fine, but pages are getting more complex, more sophisticated. It's time to take on some best practices.
You mean like taking a shower after one of these conversations with you?	
	No, like sticking to ===. It makes your code clearer and removes the potential for weird edge cases in comparisons.
Hmm. Well, ever considered just buying me out? I'd be happy to go spend my days on the beach, kicking back with a margarita in hand.	
	I didn't see that coming, I thought you'd defend your position as THE equality operator until the end. What gives?
Arguing about == versus === gets old. I mean, there are more interesting things to do in life.	
	I don't even know how to respond.
Look, here's the thing you have to deal with: people aren't going to just stop using ==. Sometimes it's really convenient. And people can use it in an educated way, taking advantage of it when it makes sense. Like the user input example—why the heck not use ==?	
	Well like I said, you never know when something is going to happen.
My new attitude is if people want to use you, great. I'm still here when they need me; and by the way, I still get a check every month no matter what they do! There's enough legacy code with == in the world—I'm never going off payroll.	


Great. If it wasn't confusing enough already, we now have two equality operators. Which one am I supposed to use?

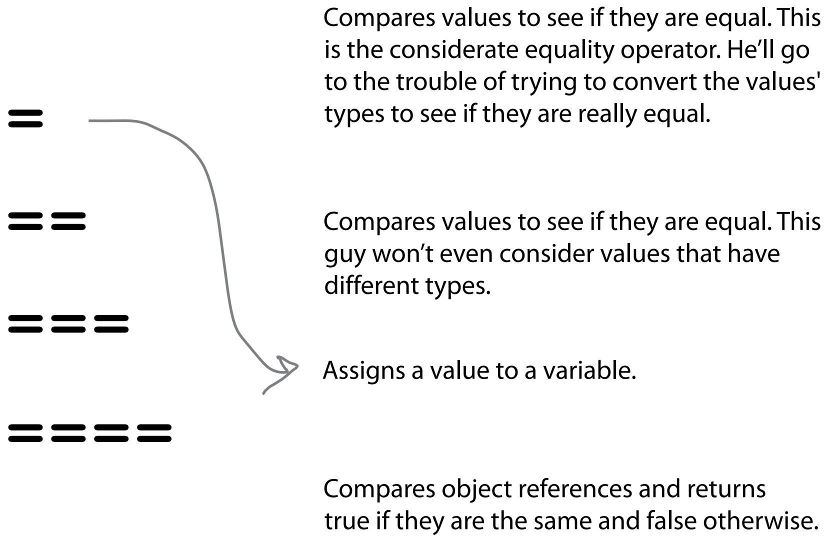
Deep breath. There's a lot of debate around this topic, and different experts will tell you different things. Here's our take: traditionally, coders have used mostly == (equality) because, well, there wasn't a great awareness of the two operators and their differences. Today, we're more educated, and for most purposes === (strict equality) works just fine and is in some ways the safer route because you know what you're getting. With ==, of course, you also know what you're getting, but with all the conversions it's hard sometimes to think through all the possibilities.

Now, there are times when == provides some nice convenience (like when you're comparing numbers to strings), and of course you should feel free to use == in those cases—especially now that, unlike many JavaScript coders, you know exactly what == does. Now that we've talked about ===, you'll see us mostly shift gears in this book and predominantly use ===, but we won't get dogmatic about it if there's a case where == makes our life easier and doesn't introduce issues.

You'll also hear developers refer to === (strict equality) as the "identity" operator.

Who Does What?

We had our descriptions of these operators all figured out, and then they got all mixed up. Can you help us figure out who does what? Be careful; we're not sure if each contender matches zero, one, or more descriptions. We've already figured one out, which is marked below.



`=`

`==`

`===`

`====`

Compares values to see if they are equal. This is the considerate equality operator. He'll go to the trouble of trying to convert the values' types to see if they are really equal.

Compares values to see if they are equal. This guy won't even consider values that have different types.

Assigns a value to a variable.

Compares object references and returns true if they are the same and false otherwise.

Solution on page 324

Even more type conversions

Conditional expressions aren't the only place you're going to see type conversion. There are a few other operators that like to convert types when they get the chance. While these conversions are meant to be a convenience for you, the coder, and often they are, it's good to understand exactly where and when they might happen. Let's take a look.

Another look at concatenation, and addition

You've probably figured out that when you use the + operator with numbers you get *addition*, and when you use it with strings you get *concatenation*. But what happens when we mix the types of +'s operands? Let's find out.

If you try to add a number and a string, JavaScript converts the number to a string and concatenates the two. Kind of the opposite of what it does with equality:

```
let addi = 3 + "4";
```

When we have a string added to a number, we get concatenation, not addition.

The addi variable is set to "34" (not 7).

```
let plusi = "4" + 3;
```

Same here...we get "43".

If you put the string first and then use the + operator with a number, the same thing happens: the number is converted to a string and the two are joined by concatenation.

What about the other arithmetic operators?

When it comes the other arithmetic operators—like multiplication, division, and subtraction—JavaScript prefers to treat those as arithmetic operations, not string operations:

```
let multi = 3 * "4";
```

Here, JavaScript converts the string "4" to the number 4 and multiplies it by 3, resulting in 12.

```
let divi = 80 / "10";
```

Here, the string "10" is converted to the number 10. Then 80 is divided by the number 10, resulting in 8.

```
let mini = "10" - 5;
```

With minus, the "10" is converted to the number 10, so we have 10 minus 5, which is 5.

there are no Dumb Questions

Q: Is + always interpreted as string concatenation when one of the operands is a string?

A: Yes. However, because + has what is called left-to-right associativity, if you have a situation like this:

```
let order = 1 + 2 + " pizzas";
```

you'll get "3 pizzas", not "12 pizzas", because, moving left to right, 1 is added to 2 first (and both are numbers), which results in 3. Next, we add 3 and a string, so 3 is converted to a string and concatenated with "pizza". To make sure you get the results you want, you can always use parentheses to force an operator to be evaluated first. This ensures you'll get 3 pizzas:

```
let order = (1 + 2) + " pizzas";
```

and this ensures you'll get 12 pizzas

```
let order = 1 + (2 + " pizzas");
```

Q: Is that it? Or are there more conversions?

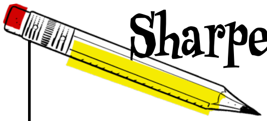
A: There are some other places where conversion happens. For instance, the unary operator – (to make a negative number) will turn –true into –1. And concatenating a boolean with a string will create a string (like true + " love" is "true love"). These cases are fairly rare, and we've personally never needed these in practice, but now you know they exist.

Q: So if I want JavaScript to convert a string into a number to add it to another number, how can I do that?

A: There's a function that does this named Number (yes, it has a uppercase N). Use it like this:

```
let num = 3 + Number("4");
```

This statement results in num being assigned the value 7. The Number function takes an argument and, if possible, creates a number from it. If the argument can't be converted to a number, Number returns...wait for it...NaN.



Sharpen your pencil

Time to test that conversion knowledge. For each expression below, write the result in the blank next to it. We've done one for you. Check your answers at the end of the chapter before you go on.

`Infinity - "1"` ______________________

`"42" + 42` "4242"

`2 + "1 1"` ______________________

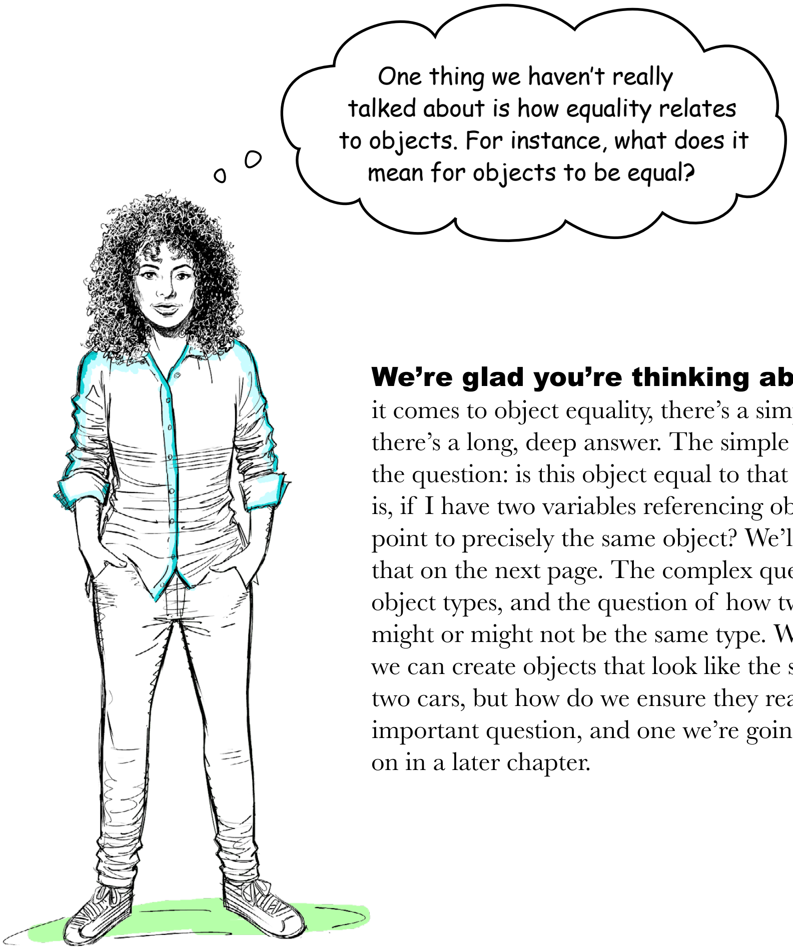
`99 + 101` ______________________

`"1" - "1"` ______________________

`console.log("Result: " + 10/2)` ______________________

`3 + " bananas " + 2 + " apples"` ______________________

→ Solution on page 325


One thing we haven't really talked about is how equality relates to objects. For instance, what does it mean for objects to be equal?

We're glad you're thinking about it. When it comes to object equality, there's a simple answer and there's a long, deep answer. The simple answer tackles the question: is this object equal to that object? That is, if I have two variables referencing objects, do they point to precisely the same object? We'll walk through that on the next page. The complex question involves object types, and the question of how two objects might or might not be the same type. We've seen that we can create objects that look like the same type, say two cars, but how do we ensure they really are? It's an important question, and one we're going to tackle head on in a later chapter.

How to determine if two objects are equal

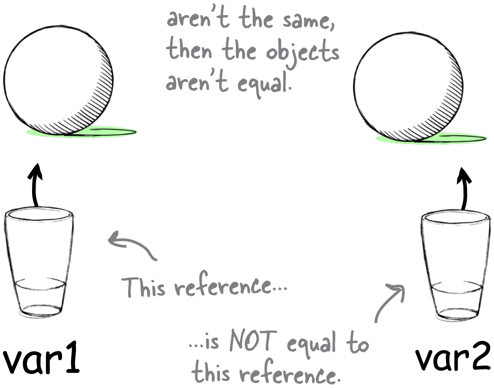
Your first question might be: are we talking about == or ===? Here's the good news: *if you're comparing two objects, it doesn't matter*! That is, if both operands are objects, then you can use either == or === because they work in exactly the same way. Here's what happens when we test two objects for equality.

When we test equality of two object variables, we compare the references to those objects.

Remember, variables hold references to objects, and so whenever we compare two objects, we're comparing object references.

```
if (var1 === var2) {
    // wow, these are the same object!
}
```

Not in this case!



Notice, it doesn't matter what's in these objects. If the references aren't the same, then the objects aren't equal.

This reference...

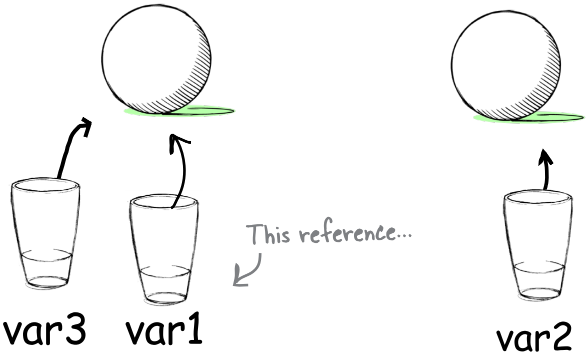
...is NOT equal to this reference. They are two different references.

Two references are equal only if they reference the same object.

The only time a test for equality between two variables containing object references returns true is when the two references point to the *same* object.

```
if (var1 === var3) {
    // wow, these are the same object!
}
```

Finally, two object references that are equal.



This reference...

...is equal to this reference. They refer the same object!

Sharpen your pencil

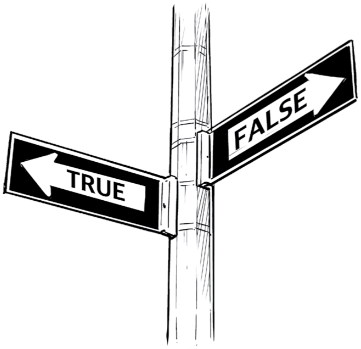
Here's a little code that helps find cars in the Earl's Autos parking lot. Trace through this code and write the values of loc1 through loc4 below.

```
function findCarInLot(car) {
    for (let i = 0; i < lot.length; i++) {
        if (car === lot[i]) {
            return i;
        }
    }
    return -1;
}
let chevy = {
    make: "Chevy",
    model: "Bel Air"
};
let taxi = {
    make: "Webville Motors",
    model: "Taxi"
};
let fiat1 = {
    make: "Fiat",
    model: "500"
};
let fiat2 = {
    make: "Fiat",
    model: "500"
};

let lot = [chevy, taxi, fiat1, fiat2];

let loc1 = findCarInLot(fiat2);    __________________
let loc2 = findCarInLot(taxi);     __________________
let loc3 = findCarInLot(chevy);    __________________
let loc4 = findCarInLot(fiat1);    __________________
```

Your answers here.

Solution on page 326

The truthy is out there...



That's right, we said truthy, not truth. We'll say falsey too. What on earth are we talking about? Well, some languages are rather precise about true and false. JavaScript, not so much. In fact, JavaScript is kind of loose about true and false. How is it loose? Well, there are values in JavaScript that aren't true or false, but that are nevertheless treated as true or false in a conditional. We call these values truthy and falsey precisely because they aren't technically true or false, but they behave like they are (again, inside a conditional).

Now, here's the secret to understanding truthy and falsey: *concentrate on knowing what is falsey, and then everything else you can consider truthy*. Let's look at some examples of using these falsey values in a conditional:

Some people write it "falsy."

```
let testThis;
if (testThis) {
   // do something
}
```

Okay, that's weird...we know this variable will be undefined in the conditional test. Does this work? Is this legal JavaScript? (Answer: yes.)

```
let element = document.getElementById("elementThatDoesntExist");
if (element) {
   // do something
}
```

Here the value of element is null. What's that going to do?

```
if (0) {
   // do another thing
}
```

We're testing 0?

Now we're doing a conditional test on an empty string. Anyone want to place bets?

```
if ("") {
   // Does code here ever get evaluated? Place your bets.
}
```

Wait, now we're using NaN in a boolean condition? What's that going to evaluate to?

```
if (NaN) {
   // Hmm, what's NaN doing in a boolean test?
}
```

What JavaScript considers falsey

Again, the secret to learning what is truthy and what is falsey is to learn what's falsey, and then consider everything else truthy.

There are five falsey values in JavaScript:

undefined is falsey.

null is falsey.

0 is falsey.

The empty string is falsey.

NaN is falsey.

To remember which values are truthy and which are falsey, just memorize the five falsey values—undefined, null, 0, "", and NaN—and remember that everything else is truthy.

Except false, which is...well, false.

So, every conditional test on the previous page evaluated to false. Did we mention every other value is truthy (except for false, of course)? Here are some examples of truthy values:

This is an array. It's not undefined, null, 0, "", or NaN. It has to be true!

```
if ([]) {
   // this will happen
}
```

This time we have an actual element object. That's not falsey either, so it's truthy.

```
let element = document.getElementById("elementThatDoesExist");
if (element) {
   // so will this
}
```

Only the number 0 is falsey; all others are truthy.

```
if (1) {
   // gonna happen
}
```

Only the empty string is falsey; all other strings are truthy.

```
let string = "mercy me";
if (string) {
   // this will happen too
}
```

Sharpen your pencil

Time for a quick lie detector test. Figure out how many lies the perp tells, and whether the perp is guilty as charged, by determining which values are truthy and which values are falsey. Check your answer at the end of the chapter before you go on—and of course, feel free to try these out in the browser yourself.

```
function lieDetectorTest() {
    let lies = 0;

    let stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    let car = {
        keysInPocket: null
    };
    if (car.keysInPocket) {
        console.log("Uh-oh, guess you stole the car!");
        lies++;
    }
    if (car.emptyGasTank) {
        console.log("You drove the car after you stole it!");
        lies++;
    }
    let foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    let yourName = " ";
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
let numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

A string with one space.

Solution on page 326

Brain Power

What do you think this code does? Do you see anything odd about the code, especially given what you know about primitive types?

```
let text = "YOU SHOULD NEVER SHOUT WHEN TYPING";
let presentableText = text.toLowerCase();
if (presentableText.length > 0) {
     alert(presentableText);
}
```

The Secret Life of Strings

Types always belong to one of two camps: they're either primitive types or objects. Primitives live out fairly simple lives, while objects keep state and have behavior (or, said another way, have properties and methods). Right?

Well, actually, while all that is true, it's not the whole story. As it turns out, strings are a little more mysterious. Check out this code:

```
let emot = "XOxxOO";
let hugs = 0;
let kisses = 0;
emot = emot.trim();
emot = emot.toUpperCase();
for (let i = 0; i < emot.length; i++) {
    if (emot.charAt(i) === "X") {
        hugs++;
    } else if (emot.charAt(i) === "O") {
        kisses++;
    }
}
```

This looks like a normal, primitive string.

Wait a sec, calling a method on a string?

And a string with a property?

More methods?

AS SEEN ON
TV

How a string can look like a primitive and an object

How can a string masquerade as both a primitive and an object? Because JavaScript supports both. That is, with JavaScript you can create a string that is a primitive, and you can also create one that is an object (which supports lots of useful string manipulation methods). Now, we've never talked about how to create a string that is an object, and in most cases you won't need to explicitly do it yourself, because the JavaScript interpreter *will create string objects for you*, as needed.

Where and why might it do that? Let's look at the life of a string:

Here we've created three primitive strings and assigned them to variables.

```
let name = "Jenny";
let phone = "867-5309";
let fact = "This is a prime number";
```

And here we're just concatenating some strings together to create another primitive string.

```
let songName = phone + "/" + name;
```

```
let index = phone.indexOf("-");
if (fact.substring(10, 15) === "prime") {
    alert(fact);
}
```

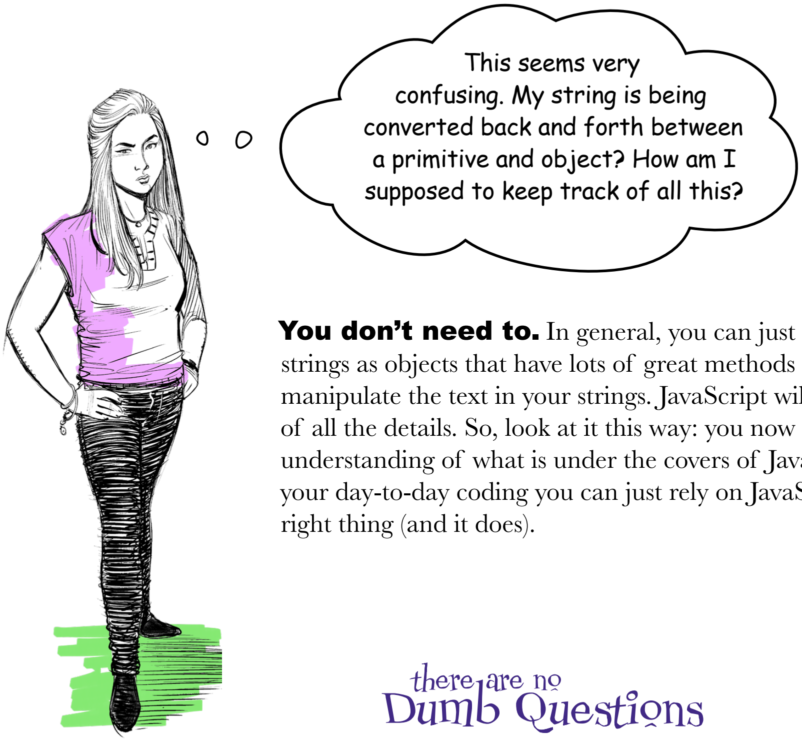
Here we're using a method. This is where, behind the scenes, JavaScript temporarily converts phone to a string object.

Same here: the fact string is temporarily converted to an object to support the substring method.

And here we're using fact again, but this time there is no need for an object, so it's back to being a boring primitive.

BORING PRIMITIVE

OBJECT WITH SUPERPOWERS


This seems very confusing. My string is being converted back and forth between a primitive and object? How am I supposed to keep track of all this?

You don't need to. In general, you can just think of your strings as objects that have lots of great methods to help you manipulate the text in your strings. JavaScript will take care of all the details. So, look at it this way: you now have a better understanding of what is under the covers of JavaScript, but in your day-to-day coding you can just rely on JavaScript to do the right thing (and it does).

there are no Dumb Questions

Q: Just making sure: do I ever have to keep track of where my string is a primitive and where it's an object?

A: Most of the time, no. The JavaScript interpreter will handle all the conversions for you. You just write your code, assuming a string supports the object properties and methods, and things will work as expected.

Q: Why does JavaScript support a string as both a primitive and an object?

A: Think about it this way: you get the efficiency of the simple string primitive type as long as you are doing basic string operations like comparison, concatenation, writing strings to the DOM, and so on. But if you need to do more sophisticated string processing, then you have the string object quickly at your disposal.

Q: Given an arbitrary string, how do I know if it is an object or a primitive?

A: A string is always a primitive unless you create it in a special way using an object constructor. We'll talk about object constructors later. And you can always use the typeof operator on your variable to see if it is of type string or object.

Q: Can other primitives act like objects?

A: Yes, numbers and booleans can also act like objects at times. However, neither of these has nearly as many useful properties as strings do, so you won't find you'll use this feature nearly as often as you do with strings. And remember, this all happens for you behind the scenes, so you don't really have to think about it much. Just use a property if you need to and let JavaScript handle the temporary conversion for you.

Q: How can I know all the methods and properties that are available for string objects?

A: That's where a good reference comes in handy. There are lots of online references that are helpful, and if you want a book, David Flanagan's *JavaScript: The Definitive Guide* has information about every string property and method in JavaScript. A search engine works pretty well too.

I hate to ask even more questions about strings, but I saw a string in some code that used backticks as delimiters, not straight or curly quotes. What is that about?

We're glad you asked. You're referring to a *template literal*, which is a fancy way of saying it's a convenient way to output variables or expressions within strings. It's super handy. As you noticed, template literals are enclosed in backticks (`` ` ``) rather than standard quotes. Within the backticks you can include standard characters, just like any string, and you can also enclose expressions within `${}`, like `${name}` or `${people * cakeslices}`, to have those values included in the string:

```
let str = `My name is ${ name }`;
```

Example of a template literal.

Let's take a moment to explore template literals, because they are so useful. After we've done that, we'll use them in the rest of the book.

How template literals work

Along with single and double quotes, you can use backticks, ``, to delimit strings. Like this:

```
let str = `What direction do you want to go in?`;
```

Here's an example of using backticks (``) to delimit a simple string.

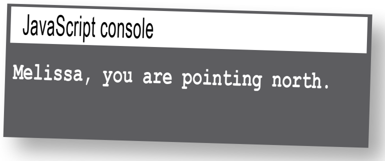
Nothing out of the ordinary happens here; the value in the variable long is just a string.

When you use a backticks, you can also include expressions in your strings. To signify an expression, we surround it by a starting `${` and a closing `}`. When the string is evaluated, any expressions are evaluated and converted to string values to output along with the rest of the string.

Like this:

```
let user = "Melissa";
let points = 10;
let direction = "north";
let nextLevel = 30;
console.log(`${user}, you are pointing ${direction}.`);
```

The values in the variables user and direction are substituted, so we see:

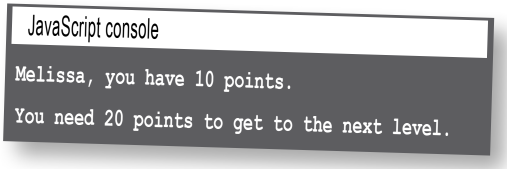


```
Melissa, you are pointing north.
```

Within template literals, what appears between the curly braces can be any expression that evaluates to or can be coerced into a string:

```
console.log(`${user}, you have ${points} points.`);
console.log(`You need ${nextLevel - points} points to get to the next level.`);
```

Here we have an expression computing the number of points left to reach the next level.



```
Melissa, you have 10 points.
You need 20 points to get to the next level.
```

You'll be seeing a lot of template literals throughout the rest of the book, as it is a readable and convenient syntax that helps us create complex strings without a bunch of concatenation operators.

A five-minute tour of string properties and methods

Given that we're in the middle of talking about strings and you've just discovered that strings also support methods, let's take a little break from discussing weirdo types and look at a few of the more common string properties and methods you might want to use. A few string methods get used over and over, and it is highly worth your time to get to know them. So, on with the tour.

A little pep talk: we could have written an entire chapter on each method and property that strings support. But that would have made this book 40 lbs and 2,000 pages long, and at this point, you really don't need it—you already get the basics of methods and objects, and all you need is a good reference if you really want to dive into the details of string processing.

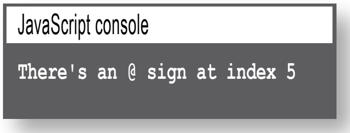
the length property

The length property holds the number of characters in the string. It's quite handy for iterating through the characters of a string.

We use the length property to iterate over each character in the string.

```
let input = "jenny@wickedlysmart.com";
for (let i = 0; i < input.length; i++) {
    if (input.charAt(i) === "@") {
        console.log(`There's an @ sign at index ${i}`);
    }
}
```

And we use the charAt method to get the character at a particular index in the string.

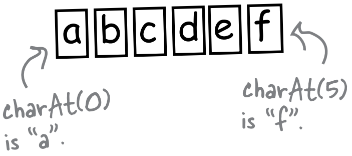


```
There's an @ sign at index 5
```

the charAt method

The charAt method takes an integer number between zero and the length in the string (minus one), and returns a string containing the single character at that position of the string. Think of a string a bit like an array, with each character at an index of the string and the indices starting at 0 (just like in an array). If you give it an index that is greater than or equal to the length of the string, charAt returns the empty string.

Note that JavaScript doesn't have a character type. So, characters are returned as new strings containing one character.



charAt(0) is "a".

charAt(5) is "f".

the indexOf method

The indexOf method takes a string as an argument and returns the index of the first character of the first occurrence of that argument in the string.

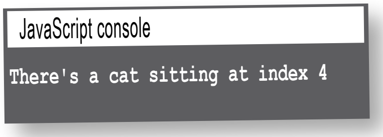
Here's the string we're going to call indexOf on.

```
let phrase = "the cat in the hat";
```

Our goal is to find the first occurrence of "cat" in phrase.

```
let index = phrase.indexOf("cat");
console.log(`There's a cat sitting at index ${index}`);
```

The index of the first cat is returned.

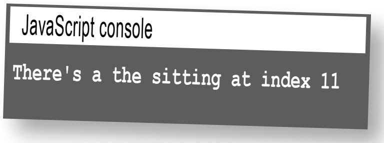


```
JavaScript console
There's a cat sitting at index 4
```

We can also add a second argument, which is the starting index for the search.

```
index = phrase.indexOf("the", 5);
console.log(`There's a the sitting at index ${index}`);
```

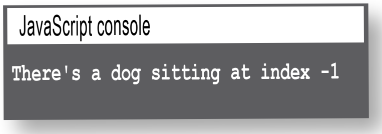
Because we're starting the search at index 5, we're skipping the first "the" and finding the second "the" at index 11.



```
JavaScript console
There's a the sitting at index 11
```

```
index = phrase.indexOf("dog");
console.log(`There's a dog sitting at index ${index}`);
```

Note: if the string can't be found, then -1 is returned as the index.



```
JavaScript console
There's a dog sitting at index -1
```

the substring method

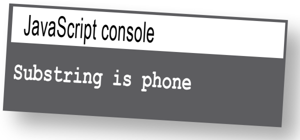
Give the substring method two indices, and it will extract and return the string contained within them.

```
let data = "name|phone|address";
let val = data.substring(5, 10);
console.log(`Substring is ${val}`);
```

Here's the string we're going to call substring on.

We'd like the string from index 5 up to (but not including) index 10 returned.

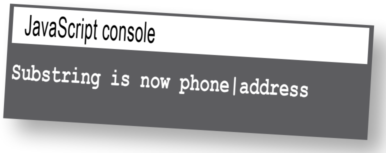
We get back a new string with the characters from index 5 to index 10.



```
Substring is phone
```

You can omit the second index, and substring will extract a string that starts at the first index and continues until the end of the original string.

```
val = data.substring(5);
console.log(`Substring is now ${val}`);
```



```
Substring is now phone|address
```

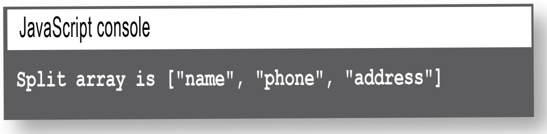
the split method

The split method takes a character that acts as a delimiter, and breaks the string into parts based on the delimiter.

```
let data = "name|phone|address";
let vals = data.split("|");
console.log("Split array is ", vals);
```

split uses the delimiter to break the original string into pieces, which are returned in an array.

Notice here we're passing two arguments to console.log, separated by a comma. This way, the vals array doesn't get converted to a string before it's displayed in the console.



```
Split array is ["name", "phone", "address"]
```

the replace method

The replace method takes two strings as arguments and replaces the first occurrence of the first argument with the second argument in the string.

```
let phone = "(201)555-1212";
```

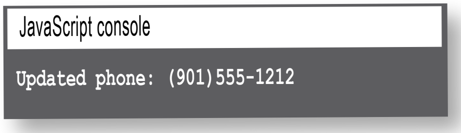
Here's a phone number. We'd like to replace the first "2" with the string "9".

```
phone = phone.replace("2", "9");
```

So, we pass the string to replace, and the string to replace it with. Note that replace returns a new string without modifying the original string.

```
console.log(`Updated phone: ${phone}`);
```

Here's the updated phone number. Notice that only the first "2" was replaced.



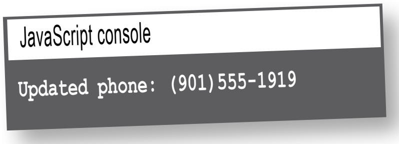
JavaScript console

```
Updated phone: (901)555-1212
```

The replaceAll method replaces all occurrences of the first argument with the second argument.

```
phone = phone.replaceAll("2", "9");
console.log(`Updated phone: ${phone}`);
```

To replace every "2" with "9", not just the first one, use replaceAll instead of replace.



JavaScript console

```
Updated phone: (901)555-1919
```

the includes method

The includes method takes one string argument and returns true if the string on which we're calling the method contains that argument. Otherwise, it returns false.

```
let phone = "(201)555-1212";
```

Here's the string we'll call includes on.

```
console.log(`Is area code "201"? ${phone.includes("(201)")}`);
```

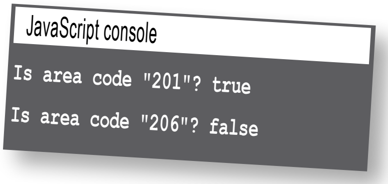
Does the string include the string "(201)"?

```
console.log(`Is area code "206"? ${phone.includes("(206)")}`);
```

Does the string include the string "(206)"?

Yes, the string phone contains "(201)", so we get true back from the first call to includes.

No, the string phone does not contain "(206)", so we get false back from the second call to includes.



JavaScript console

```
Is area code "201"? true
Is area code "206"? false
```

String Soup

`toLowerCase`
Returns a string with every uppercase character changed to lowercase.

`concat`
Joins strings together.

`replace`
Finds the first string and replaces it with another string.

`replaceAll`
Finds all strings and replaces them with another string.

`slice`
Returns a new string that has part of the original string removed.

`includes`
Returns true if the substring is contained in the string, and false otherwise.

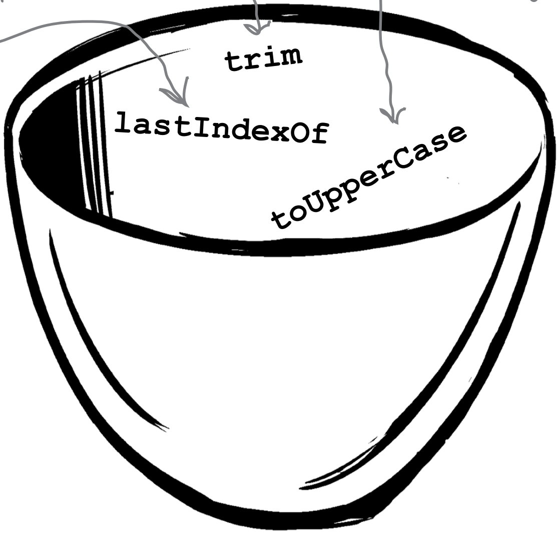
`match`
Searches for matches in a string using regular expressions.

`substring`
Returns a portion of a string.

`trim`
Removes whitespace from around the string. Handy when processing user input.

`toUpperCase`
Returns a string with every lowercase character changed to uppercase.

`lastIndexOf`
Just like indexOf, but finds the last, not the first, occurrence.



There's really no end to learning all the things you can do with strings. Here are a few more methods available to you. Just get a passing familiarity right now, and when you really need them you can look up the details.

Chair Wars
(or How Really Knowing Types Can Change Your Life)

Once upon a time in a software shop, two programmers were given the same spec and told to “build it.” The Really Annoying Project Manager forced the two coders to compete, by promising whoever delivered first gets one of those cool Aeron™ chairs all the Silicon Valley guys have. Brad, the hardcore hacker scripter, and Larry, the college grad, both knew this would be a piece of cake.

Larry, sitting in his cube, thought to himself, “What are the things this code has to *do*? It needs to make sure the string is long enough, it needs to make sure the fourth character is a dash, and it needs to make sure every other character is a number. I can use the string’s `length` property, and I know how to access its characters using the `charAt` method.”

Brad, meanwhile, kicked back at the café and asked himself the same question. He first thought, “A string is an object, and there are lots of methods I can use to help validate the phone number. I’ll brush up on those and get this implemented quickly. After all, an object is an object.” Read on to see how Brad and Larry built their programs, and for the answer to the burning question: ***who got the Aeron?***

The spec

Take a phone number of the form:

“123-4567”

and write code to accept or reject it. To be accepted the number should have seven digits, 0 through 9, with a dash in the middle.

The chair

In Larry’s cube

Larry set about writing code based on the string methods. He wrote the code in no time:

```
function validate(phoneNumber) {
    if (phoneNumber.length !== 8) {
       return false;
    }
    for (let i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.charAt(i) !== '-') {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Larry uses the length property of the string object to see how many characters it has.

He uses the charAt method to examine each character of the string.

First, he makes sure the character at index three has a dash.

Then, he makes sure all the other characters are numbers.

In Brad's cube

Brad wrote code to check for two numbers separated by a dash:

```
function validate(phoneNumber) {
   if (phoneNumber.length !== 8) {
      return false;
   }
   let first = phoneNumber.substring(0,3);
   let second = phoneNumber.substring(4);
   if (phoneNumber.charAt(3) !== "-" || isNaN(first) || isNaN(second)) {
      return false;
   }
   return true;
}
```

Brad starts just like Larry...

...but he uses his knowledge of the string methods.

He uses the substring method to create one string containing the first three characters...

...and another containing the characters from index 4 to the end of the string.

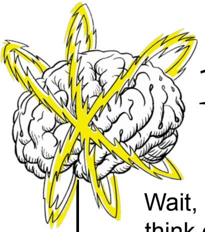
Then he tests all the conditions for a correct phone number in one conditional.

Interestingly, knowing it or not, he's depending on some type conversions here to convert a string to a number, and then making sure it's a number with isNaN. Clever!

But wait! There's been a spec change.

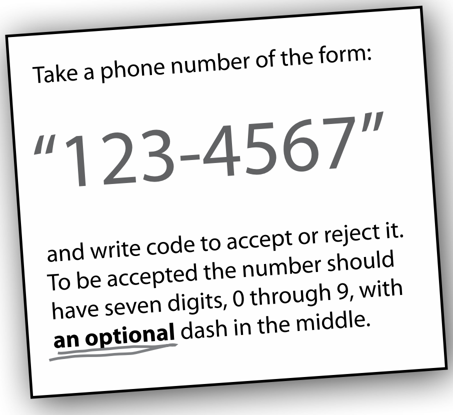
"Okay, *technically* you were first, Larry, because Brad was looking up how to use all those methods," said the Manager, "but we have to add just one tiny thing to the spec. It'll be no problem for crack programmers like you two."

"If I had a dime for every time I've heard *that* one," thought Larry, knowing that spec-change-no-problem was a fantasy. "And yet Brad looks strangely serene. What's up with that?" Still, Larry held tight to his core belief that Brad's fancy way, while cute, was just showing off, and that he'd win again in this next round and produce the code first.



Brain Power

Wait, can you think of any bugs Brad might have introduced with his use of isNaN?



Take a phone number of the form:

"123-4567"

and write code to accept or reject it. To be accepted the number should have seven digits, 0 through 9, with **an optional** dash in the middle.

What got added to the spec

Back in Larry's cube

Larry thought he could use most of his existing code; he just had to work in the edge case of the missing dash in the number. Either the number would be only seven digits, or it would be eight digits with a dash at index 3. Quickly, Larry coded the additions (which took a little testing to get right):

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
           phoneNumber.length < 7) {
       return false;
    }
    for (let i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.length === 8 &&
               phoneNumber.charAt(i) !== '-') {
              return false;
           } else if (phoneNumber.length === 7 &&
                    isNaN(phoneNumber.charAt(i))) {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Larry had to make a few additions to his logic. Not a lot of code, but it's getting a bit hard to decipher.

At the beach with Brad

Brad smiled, sipped his margarita, and quickly made his changes. He simply got the second part of the number using the length of the phone number minus four as the starting point for the substring, instead of hardcoding the starting point at a position that assumes a dash. That almost did it, but he did need to rewrite the test for the dash because it applies only when the phone number has a length of eight.

You can see Brad's updated code on the next page.

About the same number of changes as Larry, but Brad's code is still easier to read.

```
function validate(phoneNumber) {
   if (phoneNumber.length  > 8 ||
       phoneNumber.length < 7) {
       return false;
   }
   let first = phoneNumber.substring(0,3);
   let second = phoneNumber.substring(phoneNumber.length - 4);

   if (isNaN(first) || isNaN(second)) {
      return false;
   }
   if (phoneNumber.length === 8) {
      return (phoneNumber.charAt(3) === "-");
   }
   return true;
}
```

Now Brad's getting the second number using the total length of the phone number to get the starting point.

And he's validating the dash only if the number has a length of 8.

Here we're returning the result of evaluating the conditional, which will be true or false.

Brain Power

Err, we think Brad still has a bug. Can you find it?

Larry snuck in just ahead of Brad.

But the smirk on Larry's face melted when the Really Annoying Project Manager said, "Brad, your code is very readable and maintainable. Good job."

Larry shouldn't be too worried though, because as we know, good code has to do more than look pretty. The code still needs to get through QA, and we're not quite sure Brad's code works in all cases. What about you? Who do you think deserves the chair?

Brain Power

How would you rewrite Brad's code to use the split method instead?

The suspense is killing me. Who got the chair?

Amy from the second floor.

(Unbeknownst to all, the Project Manager had given the spec to *three* programmers.)

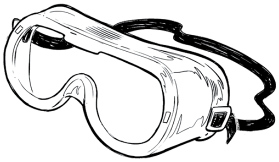
```
function validate(phoneNumber) {
    phoneNumber = phoneNumber.replace("-", "");
    return ((phoneNumber.length === 7) && !isNaN(phoneNumber));
}
```

Here's Amy's code.

Wow, a two-liner! Much cleaner and easier to understand than Brad's or Larry's code.

IN THE LABORATORY, AGAIN

The lab crew continues to probe JavaScript using the **typeof** operator, and they're uncovering some more interesting things deep within the language. In the process, they've discovered a new operator, **instanceof**. With this one, they're truly on the cutting edge. Put your lab coat and safety goggles back on and see if you can help decipher this JavaScript and the results. *Warning: this is definitely going to be the weirdest code you've seen so far.*



Here's the code. Read it, run it, alter it, massage it, see what it does...

Hmm. There are a couple of new keywords we haven't seen yet: class and constructor.

How strange. Doesn't this look a bit like a mix of a function and an object?

```
class Duck {
    constructor(sound) {
        this.sound = sound;
    }
    quack() {
        console.log(this.sound);
    }
}

let toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Hmm, "new". We've haven't seen that before. But we're guessing we should read this as "create a new Duck and assign it to the toy variable."

If it looks like an object, and walks like an object... let's test it.

Okay, and here is instanceof...

Be sure to check that your output matches ours at the end of the chapter. But just what does this all mean? Ah, we'll be getting to all that in just a couple of chapters. And, in case you didn't notice, you are well on your way to being a pretty darn advanced JavaScript coder. This is serious stuff!

JavaScript console

Put your results here. Are there any surprises?

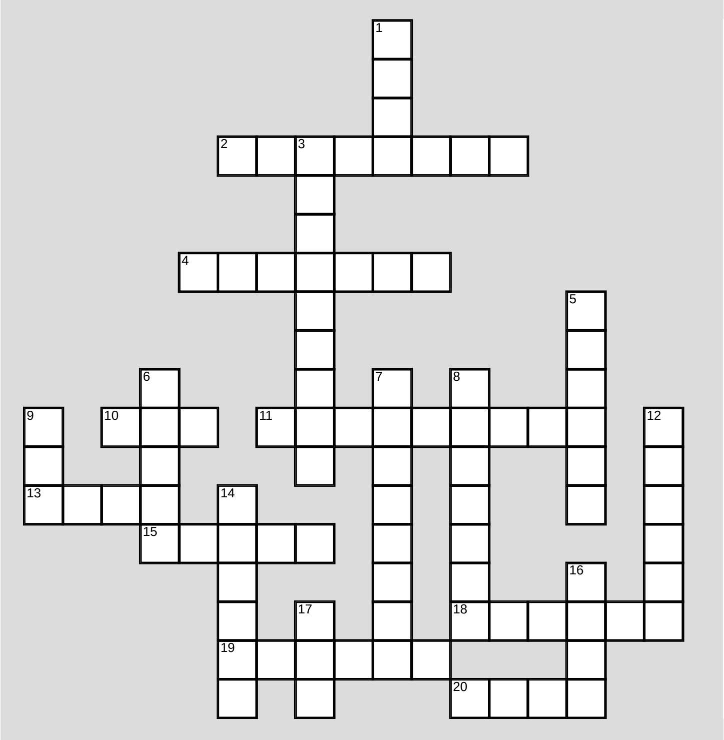
Solution on page 328

Bullet Points

- There are two groups of types in JavaScript: **primitives** and objects. Any value that isn't a primitive type is an **object**.
- The primitives are numbers, strings, booleans, null, and undefined (and BigInt and Symbol; check out the appendix). Everything else is an object.
- **undefined** means that a variable (or property or array item) hasn't yet been initialized to a value.
- **null** means "no object."
- "NaN" stands for "Not a Number," although a better way to think of **NaN** is as a number that can't be represented in JavaScript. The type of NaN is number.
- NaN never equals any other value, including itself, so to test for NaN use the function **isNaN**.
- Test two values for equality using == or ===.
- If two operands have different types, the equality operator (==) will try to convert one of the operands into another type before testing for equality.
- If two operands have different types, the strict equality operator (===) returns false.
- You can use === if you want to be sure no type conversion happens; however, sometimes the type conversion of == can come in handy.
- Type conversion is also used with other operators, like the arithmetic operators and string concatenation.
- JavaScript has five **falsey** values: undefined, null, 0, "" (the empty string), and NaN. (False is also falsey, of course.) All other values are **truthy**.
- Strings sometimes behave like objects. If you use a property or method on a primitive string, JavaScript will convert the string to an object temporarily, use the property, and then convert it back to a primitive string. This happens behind the scenes so you don't have to think about it.
- Strings have many methods that are useful for string manipulation.
- Two objects are equal only if the variables containing the object references point to the same object.

JavaScript cross

You're really expanding your JavaScript skills. Do a crossword to help it all sink in—all the answers are from this chapter.



ACROSS

2. The _______ operator tests two values to see if they're equal, after trying to convert the operands to the same type.

4. Sometimes strings masquerade as _______.

10. The only value in JavaScript that doesn't equal anything.

11. Two variables containing object references are equal only if they _______ the same object.

13. The value returned when you're expecting an object, and that object doesn't exist.

15. The _______ method is a string method that returns an array.

18. The _______ equality operator returns true only if the operands have the same type and the same value.

19. The type of Infinity is _______.

20. null == undefined

DOWN

1. The type of null in the JavaScript specification.

3. The value of a property that doesn't exist.

5. The _______ operator can be used to get the type of a value.

6. Your Fiat is parked at _______ Autos.

7. A handy way to include the value of an expression in a string is a _______ literal.

8. There are lots of handy string _______ you can use.

9. The weirdest value in the world.

12. To find a specific character at an index in a string, use the _______ method.

14. It's always 67 degrees in _______, Missouri.

16. There are _______ falsey values in JavaScript.

17. Who got the Aeron?

→ Solution on page 329

From page 277

Who am I?

A bunch of JavaScript values and party crashers, in full costume, are playing a party game: “Who am I?” They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Draw an arrow from each sentence to the name of one attendee. We’ve already guessed one of them for you.

Tonight’s attendees:

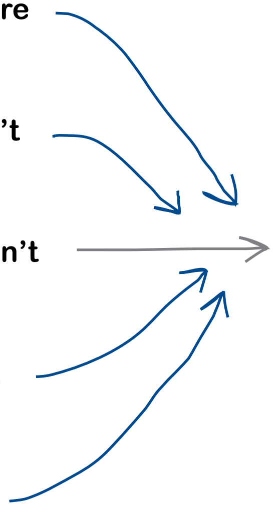
I get returned from a function when there is no return statement.

I’m the value of a variable when it hasn’t been assigned a value.

I’m the value of an array item that doesn’t exist in a sparse array.

I’m the value of a property that doesn’t exist.

I’m the value of a property that’s been deleted.



zero

empty object

null

undefined

NaN

infinity

area 51

...----...

{}

[]

From page 279

IN THE LABORATORY SOLUTION

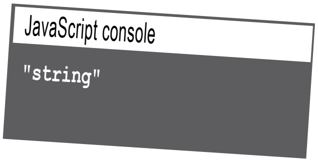
In the laboratory, we like to take things apart, look under the hood, poke and prod, hook up our diagnostic tools, and check out what is really going on. Today, we're investigating JavaScript's type system, and we've found a little diagnostic tool called **typeof** to examine variables. Put your lab coat and safety goggles on, and come on in and join us.

The **typeof** operator is built into JavaScript. You can use it to probe the type of its operand (the thing you use it to operate on). Here's an example:

```
let subject = "Just a string";

let probe = typeof subject;
console.log(probe);
```

The typeof operator takes an operand and evaluates to the type of the operand.

The type here is "string". Note that typeof uses strings to represent types, like "string", "boolean", "number", "object", "undefined", and so on.

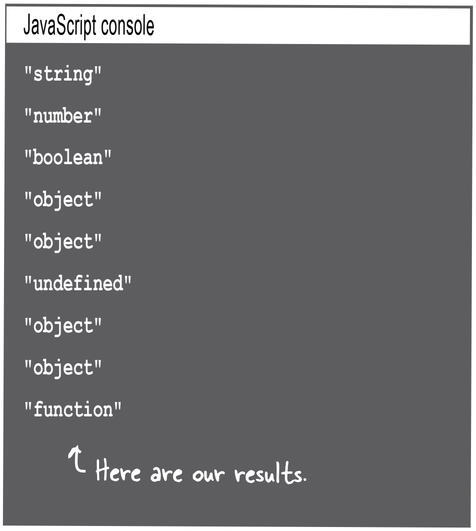


```
"string"
```

Now it's your turn. Collect the data for the following experiments:

```
let test1 = "abcdef";
let test2 = 123;
let test3 = true;
let test4 = {};
let test5 = [];
let test6;
let test7 = {"abcdef": 123};
let test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Here's the test data, and the tests.



```
"string"
"number"
"boolean"
"object"
"object"
"undefined"
"object"
"object"
"function"
```

Here are our results.

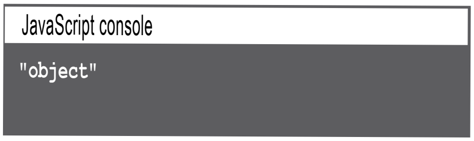
BACK IN THE LABORATORY SOLUTION

From page 280

Oops, we forgot null in our test data. Here's the missing test case:

```
let test10 = null;

console.log(typeof test10);
```

Here's our result.



```
"object"
```

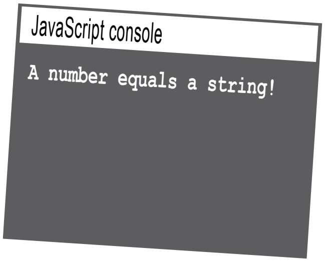
Exercise Solution

From page 284

We've been looking at some rather, um, interesting values so far in this chapter. Now, let's take a look at some interesting behavior. Try adding the code below to the <script> element in a basic web page and see what you get in the console when you load up the page. You probably won't get why yet, but see if you can take a guess about what might be going on.

```
if (99 == "99") {
    console.log("A number equals a string!");
} else {
    console.log("No way a number equals a string");
}
```

Here's what we got.



```
A number equals a string!
```

Sharpen your pencil Solution

From page 290

Write true or false below the operators == or === to represent the result of each of the following comparisons. Here's our solution.

	`==`	`===`	
`"42" == 42`	true	false	`"42" === 42`
`"0" == 0`	true	false	`"0" === 0`
`"0" == false`	true	false	`"0" === false`
`"true" == true`	false	false	`"true" === true`
`true == (1 == "1")`	true	false	`true === (1 === "1")`

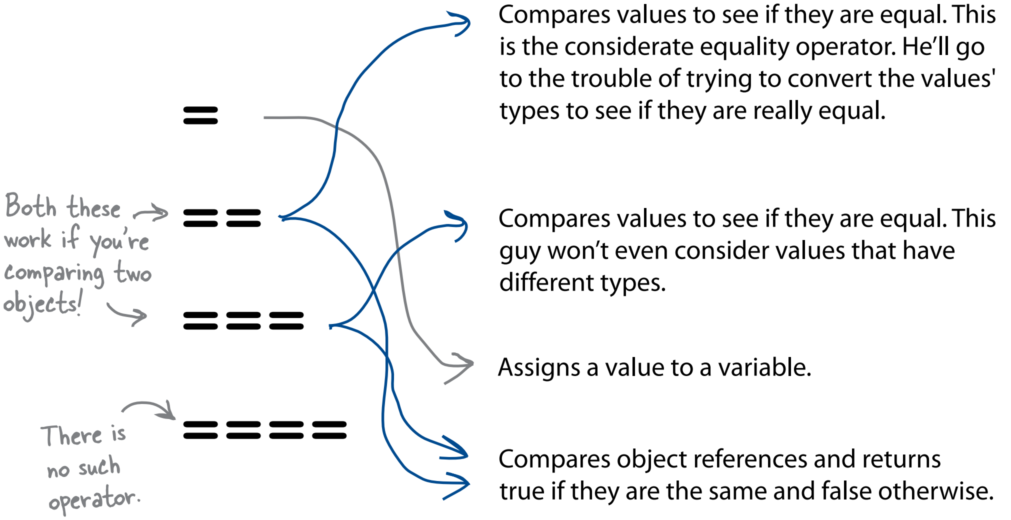
Tricky!

If you replace both == with ===, then the result is false.

Who Does What? Solution

From page 295

We had our descriptions of these operators all figured out, and then they got all mixed up. Can you help us figure out who does what? Be careful; we're not sure if each contender matches zero, one, or more descriptions. Here's our solution.



Compares values to see if they are equal. This is the considerate equality operator. He'll go to the trouble of trying to convert the values' types to see if they are really equal.

Compares values to see if they are equal. This guy won't even consider values that have different types.

Assigns a value to a variable.

Compares object references and returns true if they are the same and false otherwise.

Both these work if you're comparing two objects!

There is no such operator.

Sharpen your pencil Solution

From page 297

Time to test that conversion knowledge. For each expression below, write the result in the blank next to it. Here's our solution.

`Infinity - "1"` Infinity

"1" is converted to 1, and Infinity - 1 is Infinity.

`"42" + 42` "4242"

`2 + "1 1"` "21 1"

`99 + 101` 200

`"1" - "1"` 0

Both strings are converted to 1, and 1 - 1 is 0.

`console.log("Result: " + 10/2)` "Result: 5"

10 / 2 happens first, and the result is concatenated to the string "Result: ".

`3 + " bananas " + 2 + " apples"` "3 bananas 2 apples"

Each + is concatenation because for both, one operand is a string.

Sharpen your pencil Solution

From page 303

Time for a quick lie detector test. Figure out how many lies the perp tells, and whether the perp is guilty as charged, by determining which values are truthy and which values are falsey. Here's our solution. Did you try these out in the browser yourself?

```
function lieDetectorTest() {
    let lies = 0;

    let stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    let car = {
        keysInPocket: null
    };
    if (car.keysInPocket) {
        console.log("Uh-oh, guess you stole the car!");
        lies++;
    }
    if (car.emptyGasTank) {
        console.log("You drove the car after you stole it!");
        lies++;
    }
    let foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    let yourName = " ";
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
let numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

Any object is truthy, even an empty one.

This perp didn't steal the car because the value of the keysInPocket property is null, which is falsey.

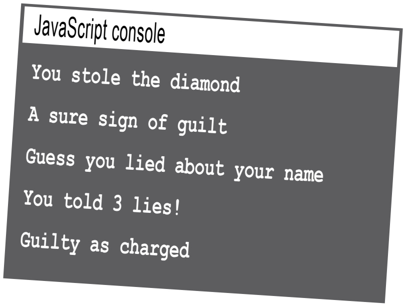
And the perp didn't drive the car either, because the emptyGasTank property is undefined, which is falsey.

But [ ] (an empty array) is truthy, so the perp was caught on the scene.

There is no item in the array, so the array item at 0 is undefined, which is falsey. Hmm, the perp must have hidden the stash already.

A string with one space.

Any nonempty string is truthy, even if it just has one space!



```
You stole the diamond
A sure sign of guilt
Guess you lied about your name
You told 3 lies!
Guilty as charged
```

The number of lies is 3, so we think the perp is guilty.

Sharpen your pencil Solution

From page 300

Here's a little code that helps find cars in the Earl's Autos parking lot. Trace through this code and write the values of loc1 through loc4 below.

```
function findCarInLot(car) {
    for (let i = 0; i < lot.length; i++) {
        if (car === lot[i]) {
            return i;
        }
    }
    return -1;
}
let chevy = {
    make: "Chevy",
    model: "Bel Air"
};
let taxi = {
    make: "Webville Motors",
    model: "Taxi"
};
let fiat1 = {
    make: "Fiat",
    model: "500"
};
let fiat2 = {
    make: "Fiat",
    model: "500"
};

let lot = [chevy, taxi, fiat1, fiat2];
```

Here are our answers.

```
let loc1 = findCarInLot(fiat2);    3
let loc2 = findCarInLot(taxi);     1
let loc3 = findCarInLot(chevy);    0
let loc4 = findCarInLot(fiat1);    2
```

From page 318

IN THE LABORATORY, AGAIN

SOLUTION

The lab crew continues to probe JavaScript using the **typeof** operator, and they're uncovering some more interesting things deep within the language. In the process, they've discovered a new operator, **instanceof**. With this one, they're truly on the cutting edge. Put your lab coat and safety goggles back on and see if you can help decipher this JavaScript and the results.

Warning: this is definitely going to be the weirdest code you've seen so far.

Here's the code. Read it, run it, alter it, massage it, see what it does...

Hmm. There are a couple of new keywords we haven't seen yet: class and constructor.

How strange. Doesn't this look a bit like a mix of a function and an object?

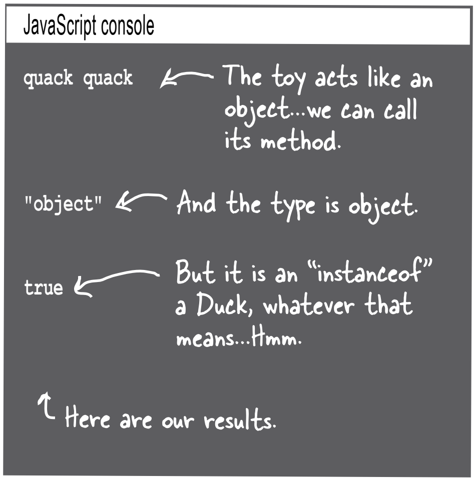
```
class Duck {
    constructor(sound) {
        this.sound = sound;
    }
    quack() {
        console.log(this.sound);
    }
}

let toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Hmm, "new". We've haven't seen that before. But we're guessing we should read this as "create a new Duck and assign it to the toy variable."

If it looks like an object, and walks like an object... let's test it.

Okay, and here is instanceof...

Just what does this all mean? Ah, we'll be getting to all that in just a few chapters. And, in case you didn't notice, you are well on your way to being a pretty darn advanced JavaScript coder. This is serious stuff!


JavaScript console

```
quack quack
"object"
true
```

The toy acts like an object...we can call its method.

And the type is object.

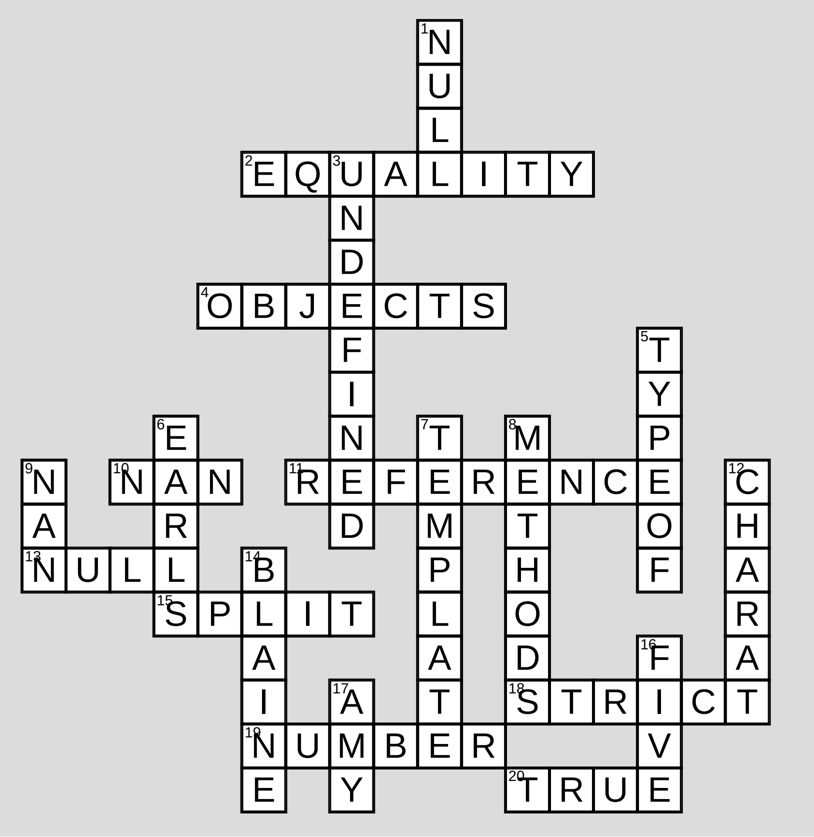
But it is an "instanceof" a Duck, whatever that means...Hmm.

Here are our results.

JavaScript cross Solution

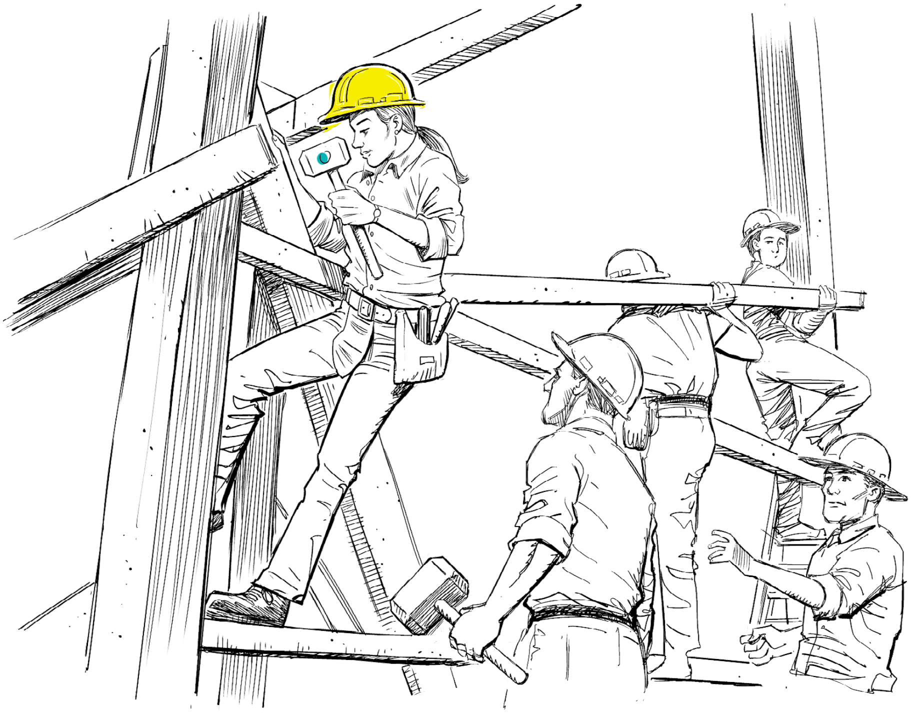
From page 320

You're really expanding your JavaScript skills. Do a crossword to help it all sink in—all the answers are from this chapter. Here's our solution.


1 NULL
2 EQUALITY
3 UNDEFINED
4 OBJECTS
5 TYPEOF
6 EARL
7 TEMPLATE
8 METHOD
9 NAN
10 NAN
11 REFERENCE
12 CHARAT
13 NULL
14 BLAINE
15 SPLIT
16 FIVE
17 AMY
18 STRICT
19 NUMBER
20 TRUE

8 bringing it all together

Building an app

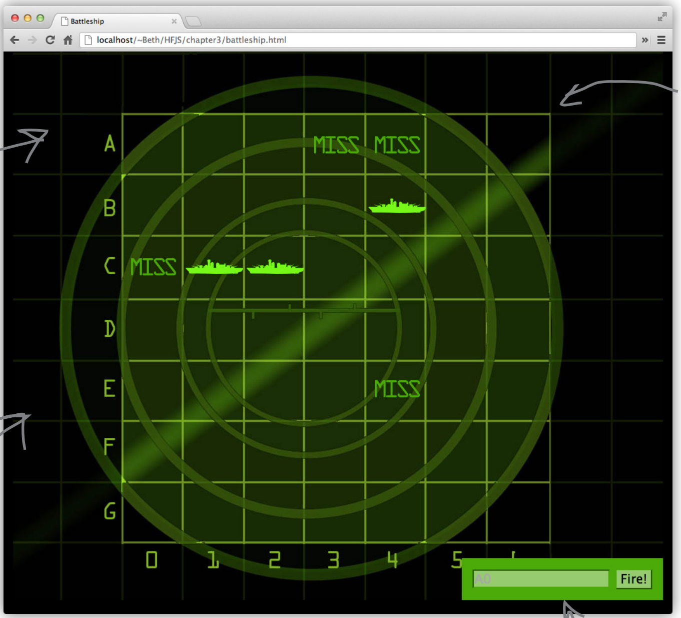


Put on your toolbelt. That is, the toolbelt with all your new coding skills, your knowledge of the DOM, and even some HTML and CSS. We're going to bring everything together in this chapter to create our first true **web application**. No more **silly toy games** with one battleship and a single row of hiding places. In this chapter, we're building the **entire experience**: a nice big game board, multiple ships, and user input right in the web page. We're going to create the page structure for the game with HTML, visually style the game with CSS, and write JavaScript to code the game's behavior. Get ready: this is an all-out, pedal-to-the-metal development chapter where we're going to lay down some serious code.

This time, let's build a REAL Battleship game

Sure, you can feel good because back in Chapter 2 you built a nice little Battleship game from scratch. But let's admit it: that was a bit of a *toy* game—it worked, it was playable, but it wasn't exactly a game you'd impress your friends with, or use to raise your first round of venture capital. To really impress, you'll need a visual game board, snazzy battleship graphics, and a way for players to enter their moves right in the game (rather than a generic browser dialog box). You'll also want to improve the previous version by supporting all three ships.

In other words, you'll want to create something like this:

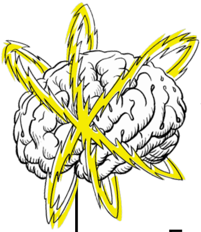


Highly visual game board right in the web page, complete with a grid.

Your hits and misses are placed right on the game board as you take your shots.

And now we've got all three ships ducking your shots.

Enter your guesses right in the page.



Brain Power

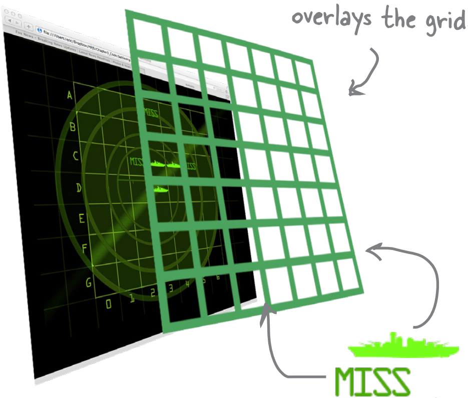
Forget JavaScript for a minute...look at the Battleship mockup above. If you focus on the structure and visual represention of the page, how would you create it using HTML and CSS?

Stepping back...to HTML and CSS

To create a modern, interactive web page, or *app*, you need to work with three technologies: HTML, CSS, and JavaScript. You already know the mantra "HTML is for structure, CSS is for style, and JavaScript is for behavior." But rather than just stating it, in this chapter we're going to fully embody it. And we're going to begin with the HTML and CSS.

Our first goal is going to be to reproduce the look of the game board on the previous page. But not *just* reproduce it; we need to implement the game board so it has a structure we can use in JavaScript to take player input and place hits, misses, and messages on the page.

To pull that off we're going to use an image in the background to give us the slick grid over a radar look, and then we'll lay a more functional HTML table over that so we can place things (like ships) on top of it. We'll also use an HTML form to get the player input.


We'll place an image in the background of the page that depicts the grid of the game.

And then add an HTML table that overlays the grid.

Then we can place the ship or MISS graphics in the table cells as needed.

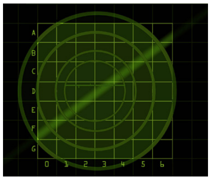
So, let's build this game. We're going to take a step back and spend a few pages on the crucial HTML and CSS, but once we have that in place, we'll be ready for the JavaScript.

GET YOUR Battleship Toolkit

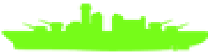
Here's a toolkit to get you started on this new version of Battleship:

INVENTORY includes...

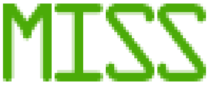
board.jpg



ship.png



miss.png



This toolkit contains three images:

- "board.jpg", which is the main background board for the game, including the grid.
- "ship.png", which is a small ship for placement on the board. Notice that it is a PNG image with transparency, so it will lay right on top of the background.
- "miss.png", which is also meant to be placed on the board.

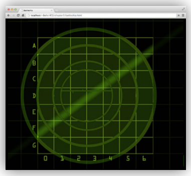
True to the original game, when we hit a ship we place a ship graphic in the corresponding cell, and when we miss we place a MISS graphic there.

Download everything you need for the game at https://wickedlysmart.com/hfjs.

Creating the HTML page: the Big Picture

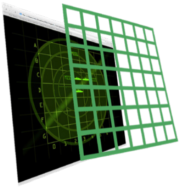
Here's the plan of attack for creating the Battleship HTML page:

1. First, we'll concentrate on the background of the game. We'll set the background color to black and place the radar grid image in the page.



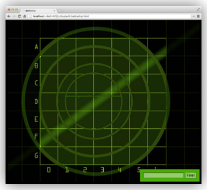
We're placing an image in the background to give the game its cool, green phosphorus radar feel.

2. Next, we'll create an HTML table and lay it on top of the background image. Each cell in the table will represent a board cell in the game.



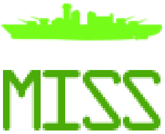
An HTML table on top of the background creates a game board for the game to play out in.

3. Then, we'll add an HTML form where players can enter their guesses, like "A4". We'll also add an area to display messages, like "You sank my battleship!"



We add an HTML form for player input.

4. Finally, we'll figure out how to use the table to place the images of a battleship (for a hit) and a MISS (for a miss) on the board.



We'll use these images and place them into the table as needed.

Step 1: The basic HTML

Let's get started! First, we need an HTML page. We're going to start by creating a simple HTML5-compliant page; we'll also add some styling for the background image. In the page, we're going to place a `<body>` element that contains a single `<div>` element. This `<div>` element is going to hold the game grid.

Go ahead and check out the next page, which contains our starter HTML and CSS.

Relax

If you're feeling a bit rusty on your HTML and CSS, *Head First HTML and CSS* was written to be the companion to this book.

Head First HTML and CSS

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Battleship</title>
    <style>
      body {
        background-color: black;
      }

      div#board {
        position: relative;
        width: 1024px;
        height: 863px;
        margin: auto;
        background: url("board.jpg") no-repeat;
      }
    </style>
  </head>
  <body>
    <div id="board">

    </div>
    <script src="battleship.js"></script>
  </body>
</html>
```

Just a regular HTML page.

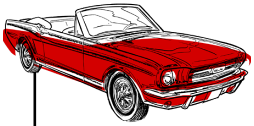
We want the background of the page to be black.

We want the game board to stay in the middle of the page, so we're setting the width to 1024px (the width of the game board) and the margin to auto.

Here's where we add the "board.jpg" image to the page, as the background of the "board" <div> element. We're positioning this <div> relative, so that we can position the table we add in the next step relative to this <div>.

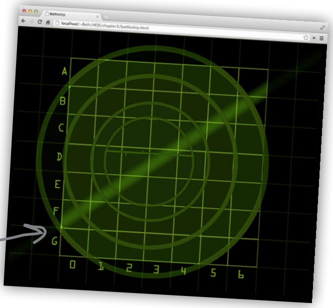
We're going to put the table for the game board and the form for getting user input here.

We'll put our code in the file "battleship.js". Go ahead and create a blank file for that.



Test Drive

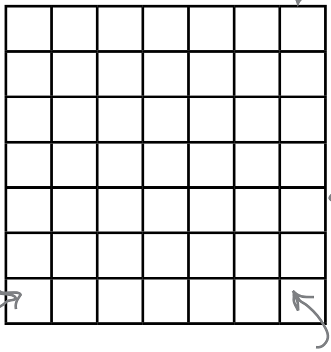
Go ahead and enter the code above into a file called "battleship.html" (or download it with the rest of the code for the book from https://wickedlysmart.com/hfjs), and then load it in your browser. Our test run is below.



Here's what the web page looks like so far...

Step 2: Creating the table

Next up is the table. The table will overlay the visual grid in the background image and provide the area to place the hit and miss graphics when you play the game. Each cell (or, if you remember your HTML, each `<td>` element) is going to sit right on top of a cell in the background image. Now here is the trick: we'll give each cell its own id, so we can manipulate it later with CSS and JavaScript. Let's check out how we're going to create these ids and add the HTML for the table:



Each cell of the grid corresponds to a <td> in the table.

Each id represents a cell's location in the grid. So, the top-left cell gets the id "00" and the bottom-right cell gets the id "66".

In the game, the rows are represented with letters (A, B, C, etc.), but for our ids we're going to replace each letter with its corresponding number (0, 1, 2, etc.). So, the first row of the grid is row 0 (for A), and the last one is 6 (for G).

This cell has id="60"

This is the cell with id "66".

Here's the HTML for the table. Go ahead and add this between the `<div>` tags:

```
<div id="board">
  <table>
    <tr>
      <td id="00"></td><td id="01"></td><td id="02"></td><td id="03">
</td><td id="04"></td> <td id="05"></td><td id="06"></td>
    </tr>
    <tr>
      <td id="10"></td><td id="11"></td><td id="12"></td><td id="13"></td>
<td id="14"></td> <td id="15"></td><td id="16"></td>
    </tr>
    ...
    <tr>
      <td id="60"></td><td id="61"></td><td id="62"></td><td id="63"></td>
<td id="64"></td><td id="65"></td><td id="66"></td>
    </tr>
  </table>
</div>
```

We're nesting the table inside the "board" <div>.

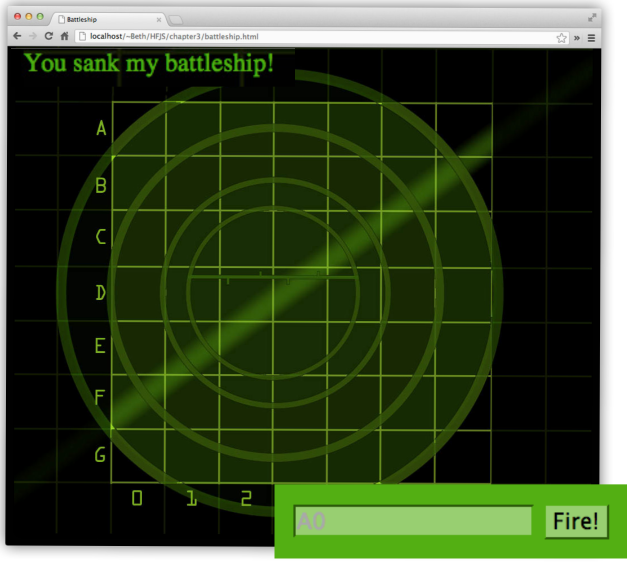
Make sure each <td> gets the correct id corresponding to its row and column in the grid.

We've left out a few rows to save some trees, but we're sure you can fill these in on your own.

Step 3: Player interaction

Okay, now we need an HTML element to enter guesses (like “A0” or “E4”), and an element to display messages to the player (like “You sank my battleship!”). We’ll use a `<form>` with a text `<input>` for the player to submit guesses, and a `<div>` to create an area where we can message the player:

We’ll notify players when they’ve sunk battleships with a message up in the top-left corner.



You sank my battleship!

And here’s where players can enter their guesses.

```
<div id="board">
  <div id="messageArea"></div>
  <table>
    ...
  </table>
  <form>
    <input type="text" id="guessInput" placeholder="A0">
    <input type="button" id="fireButton" value="Fire!">
  </form>
</div>
```

The “messageArea” <div> will be used to display messages from code.

The <form> has two inputs: one for the guess (a text input) and one for the button. Note the ids on these elements. We’ll need them later when we write the code to get the player’s guess.

Notice that the “messageArea” <div>, the <table>, and the <form> are all nested within the “board” <div>. This is important for the CSS on the next page.

Adding some more style

If you load the page now (go ahead, give it a try), most of the elements are going to be in the wrong places and the wrong size. We need to provide some CSS to put everything in the right place and make sure all the elements, like the table cells, are the right size to match up with the game board image.

To get the elements into the right places, we're going to use CSS positioning to lay everything out. We've relatively positioned the "board" `<div>` element, so we can now position the message area, table, and form at specific places within the "board" `<div>` to get them to display exactly where we want them.

Let's start with the "messageArea" `<div>`. It's nested inside the "board" `<div>`, and we want to position it at the top-left corner of the game board:

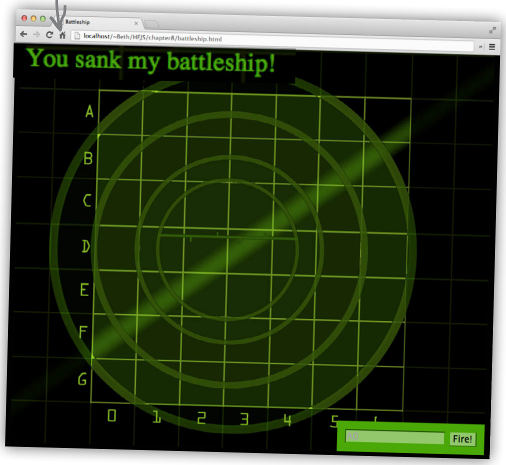
```
body {
    background-color: black;
}
div#board {
    position: relative;
    width: 1024px;
    height: 863px;
    margin: auto;
    background: url("board.jpg") no-repeat;
}
div#messageArea {
    position: absolute;
    top: 0px;
    left: 0px;
    color: rgb(83, 175, 19);
}
```

The "board" <div> is positioned relative, so everything nested within this <div> can be positioned relative to it.

We're positioning the message area at the top left of the board.

The "messageArea" <div> is nested inside the "board" <div>, so its position is specified relative to the "board" <div>. It will be positioned 0px from the top and 0px from the left of the top-left corner of the "board" <div>.

We want the message area to be positioned at the top-left corner of the game board.


You sank my battleship!

Bullet Points

- "position: relative" positions an element at its normal location in the flow of the page.
- "position: absolute" positions an element based on the position of its most closely positioned parent.
- The top and left properties can be used to specify the number of pixels to offset a positioned element from its default position.

We can also position the table and the form within the "board" `<div>` , again using absolute positions to get these elements precisely where we want them. Here's the rest of the CSS:

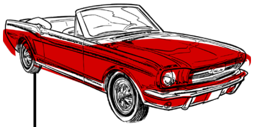
```
body {
    background-color: black;
}
div#board {
    position: relative;
    width: 1024px;
    height: 863px;
    margin: auto;
    background: url("board.jpg") no-repeat;
}
div#messageArea {
    position: absolute;
    top: 0px;
    left: 0px;
    color: rgb(83, 175, 19);
}
table {
    position: absolute;
    left: 173px;
    top: 98px;
    border-spacing: 0px;
}
td {
    width: 94px;
    height: 94px;
}
form {
    position: absolute;
    bottom: 0px;
    right: 0px;
    padding: 15px;
    background-color: rgb(83, 175, 19);
}
form input {
    background-color: rgb(152, 207, 113);
    border-color: rgb(83, 175, 19);
    font-size: 1em;
}
```

We position the <table> 173 pixels from the left of the board and 98 pixels from the top, so it aligns with the grid in the background image.

Each <td> gets a specific width and height so that the cells of the <table> match up with the cells of the grid.

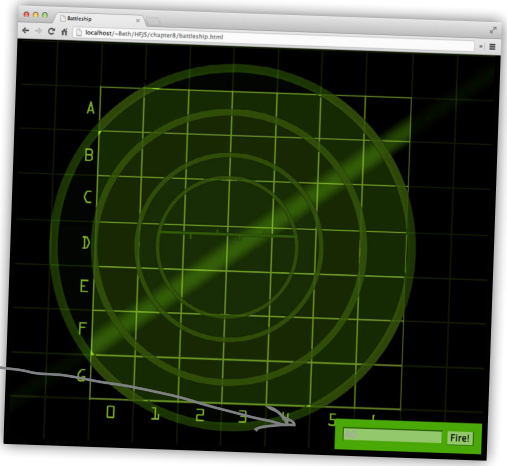
We're placing the <form> at the bottom right of the board. It obscures the bottom-right numbers a bit, but that's okay (you know what they are). We're also giving the <form> a nice green color to match the background image.

And finally, a bit of styling on the two <input> elements so they fit in with the game theme, and we're done!



Test Drive

It's time for another game checkpoint. Get all the HTML and CSS entered into your HTML file and then reload the page in your browser. Here's what you should see:

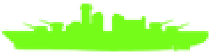


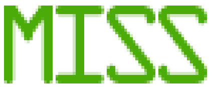
Even though you can't see it (because it's invisible), the table is sitting right on top of the grid.

The form input is ready to take your guesses, although nothing will happen until we write some code.

Step 4: Placing the hits and misses

ship.png





miss.png

The game board is looking great, don't you think? However, we still need to figure out how to visually add hits and misses to the board—that is, how to add either a "ship.png" image or a "miss.png" image to the appropriate spot on the board for each guess. Right now, we're only going to worry about how to craft the right markup or style to do this; later, we'll use the same technique in code.

So how do we get a "ship.png" image or a "miss.png" image on the board? A straightforward way is to add the appropriate image to the background of a `<td>` element using CSS. Let's try that by creating two classes, one named "hit" and the other "miss". We'll use the `background` CSS property with these images, so an element styled with the "hit" class will have the "ship.png" image in its background and an element styled with the "miss" class will have the "miss.png" image in its background. Like this:

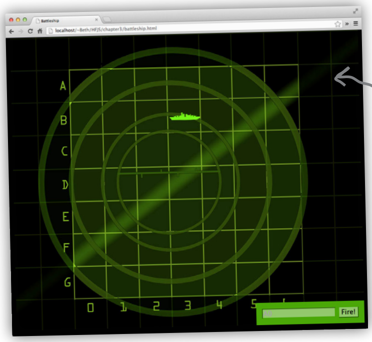
If an element is in the hit class, it gets the "ship.png" image. If the element is in the miss class, it gets the "miss.png" image in its background.

```
.hit {
     background: url("ship.png") no-repeat center center;
 }
.miss {
     background: url("miss.png") no-repeat center center;
 }
```

Each CSS rule places a single centered image in the selected element.

Using the hit and miss classes

Make sure you've added the hit and miss class definitions to your CSS. You may be wondering how we're going to use these classes. Let's do a little experiment right now to demonstrate. Imagine you have a ship hidden at B3, B4, and B5, and the user guesses "B3"—a hit! So, you need to place a "ship.png" image at B3. Here's how you can do that. First, convert the "B" into a number, 1 (since A is 0, B is 1, and so on), and find the `<td>` with the id "13" in your table. Then add the class "hit" to that `<td>`, like this:



Here we've added the "hit" class to the <td>.

```
<tr>
<td id="10"></td> <td id="11"></td> <td id="12"></td> <td id="13" class="hit"></td>
<td id="14"></td> <td id="15"></td> <td id="16"></td>
</tr>
```

Make sure you've added the hit and miss classes from the previous page to your CSS.

Now when you reload the page, you'll see a battleship at location "B3" in the game board.

What we see when we add the class "hit" to the element with id "13".

Sharpen your pencil

Before we write the code that's going to place hits and misses on the game board, get a little more practice to see how the CSS works. Manually play the game by adding the "hit" and "miss" classes into your markup, as dictated by the player's moves below. Imagine you've got ships at these locations:

```
Ship 1: A6, B6, C6
Ship 2: C4, D4, E4
Ship 3: B0, B1, B2
```

Remember, you'll need to convert the letters to numbers, with A = 0, ... G = 6.

and here are the player's guesses:

```
A0, D4, F5, B2, C5, C6
```

Be sure to check your solution at the end of the chapter before you go on.

When you're done, remove any classes that you've added to your <td> elements so you'll have an empty board to use when we start coding.

Solution on page 390

there are no Dumb Questions

Q: I didn't know it was okay to use a string of numbers for the id attributes in our table.

A: As of HTML5, you are allowed to use all numbers as an element id. As long as there are no spaces in the id value, it's fine. And for the Battleship application, using numbers for each id works perfectly as a way to keep track of each table position, so we can access the element at that position quickly and easily.

Q: So just to make sure I understand, we're using each <td> element as a cell in the game board, and we'll mark a cell as being a hit or a miss with the class attribute?

A: Right. There are a few pieces here: we have a background image grid that is just for eye candy, we have a transparent HTML table overlaying that, and we use the classes "hit" and "miss" to put an image in the background of each table cell when needed. This last part will all be done from code, when we're going to dynamically add the class to an element.

Q: It sounds like we're going to need to convert letters, as in "A6", to numbers, so we get "06". Will JavaScript do this automatically for us?

A: No, we're going to have to do that ourselves, but we have an easy way to do it—we're going to use what you know about arrays to do a quick conversion… stay tuned.

Q: I'm not sure I completely remember how CSS positioning works.

A: Positioning allows you to specify an exact position for an element. If an element is positioned "relative", then the element is positioned based on its normal location in the flow of the page. If an element is positioned "absolute", then that element is positioned at a specific location, relative to its most closely positioned parent. Sometimes that's the entire page, in which case the position you specify could be its top-left position based on the corner of the web browser. In our case, we're positioning the table and message area elements absolutely, but in relation to the game board (because the board is the most closely positioned parent of the table and the message area).

If you need a more in-depth refresher on CSS positioning, check out Chapter 11 of *Head First HTML and CSS*.

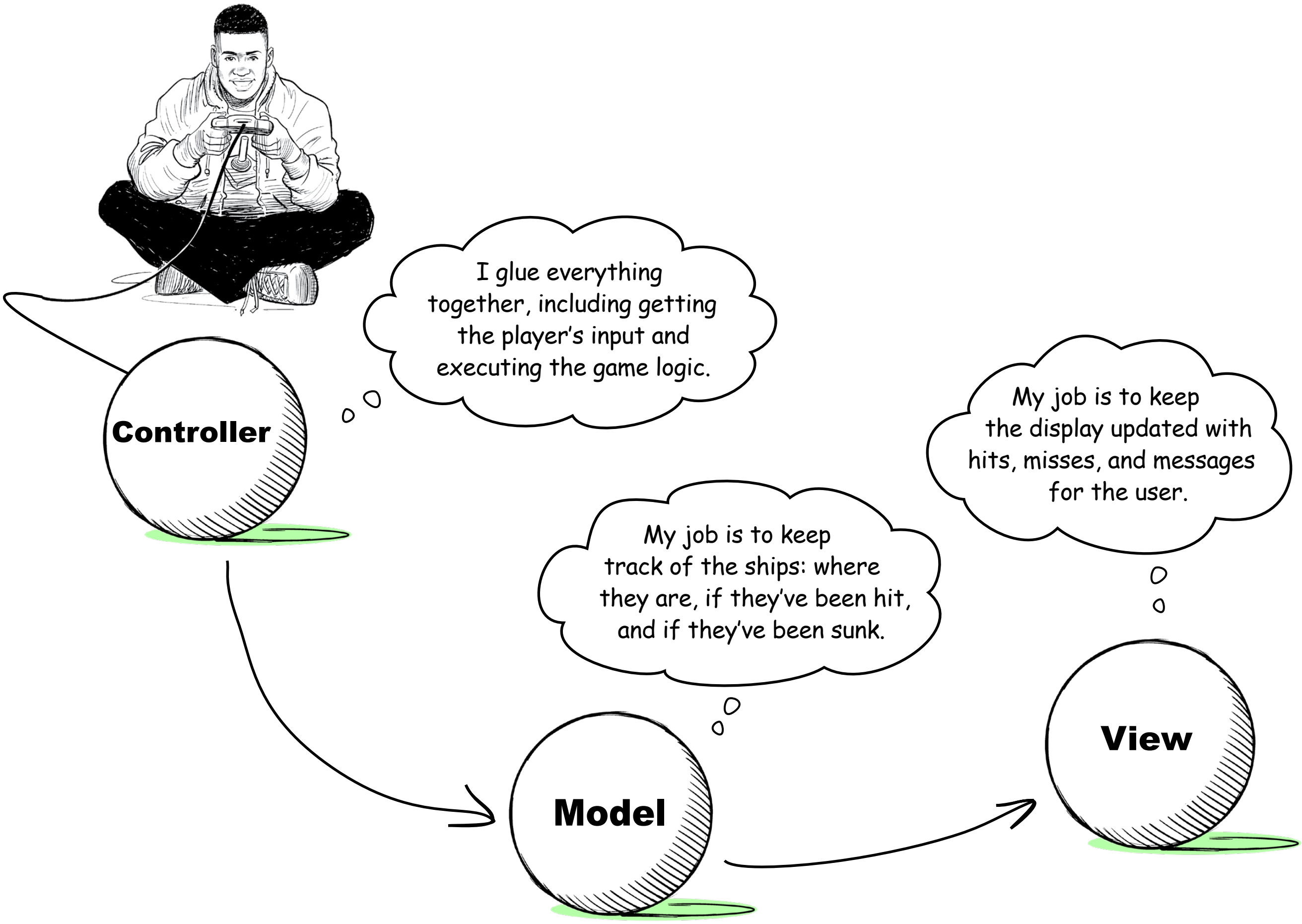
Q: When I learned about the HTML <form> element, I was taught there is an action attribute that submits the form. Why don't we have one?

A: We don't need the action attribute in the <form> because we're not submitting the form to a server-side application. For this game, we're going to be handling everything in the browser, using code. So, instead of submitting the form, we're going to implement an event handler to be notified when the form button is clicked, and when that happens, we'll handle everything in our code, including getting the user's input from the form. Notice that the type of the form button is "button", not "submit", like you might be used to seeing if you've implemented forms that submit data to a PHP program or another kind of program that runs on the server. It's a good question; more on this later in the chapter.

Designing the game

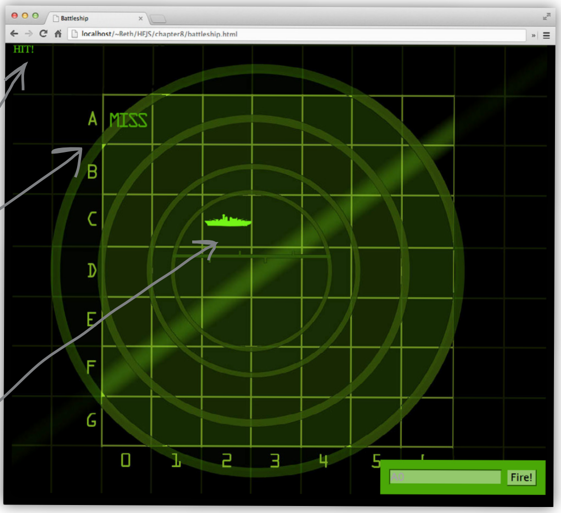
With the HTML and CSS out of the way, let's get to the real game design. Back in Chapter 2, we hadn't covered functions or objects or encapsulation or talked about object-oriented design, so when we built the first version of the Battleship game, we used a procedural design—that is, we designed the game as a series of steps, with some decision logic and iteration mixed in. You also hadn't learned about the DOM, so the game wasn't very interactive. This time around, we're going to organize the game into a set of objects, each with its own responsibilities, and we're going to use the DOM to interact with the user. You'll see how this design makes approaching the problem a lot more straightforward.

Let's first introduce the objects we're going to design and implement. There are three: the *model*, which will hold the state of the game, like where each ship is located and where it's been hit; the *view*, which is responsible for updating the display; and the *controller*, which glues everything together by handling the user input, making sure the game logic gets executed, and determining when the game is over.


I glue everything together, including getting the player's input and executing the game logic.
Controller
My job is to keep the display updated with hits, misses, and messages for the user.
My job is to keep track of the ships: where they are, if they've been hit, and if they've been sunk.
View
Model

Exercise

It's time for some object design. We're going to start with the view object. Now remember, the view object is responsible for updating the view. Take a look at the view below and see if you can determine the methods we want the view object to implement. Write down the declarations for these methods (just the declarations; we'll code the bodies of the methods in a bit), along with a comment or two about what each does. We've done one for you. Check your answers before moving on.



Here's a message. Messages will be things like "HIT!", "You missed." and "You sank my battleship!"

Here the display has a MISS placed on the grid.

And here the display has a ship placed on the grid.

```
let view = {

   // this method takes a string message and displays it
   // in the message display area
   displayMessage(msg) {
          // code to be supplied in a bit!
   }
```

Your methods go here!

```
};
```

Solution on page 391

Implementing the view

If not, shame on you. Do it now!

If you checked our solution to the previous exercise, you've seen that we've broken the view into three separate methods: `displayMessage`, `displayHit`, and `displayMiss`. Now, there is no one right answer. For instance, you might have just two methods, `displayMessage` and `displayPlayerGuess`, and pass an argument into `displayPlayerGuess` that indicates if the player's guess was a hit or a miss. That is a perfectly reasonable design. But we're sticking with our design for now...so let's think through how to implement the first method, `displayMessage`:

Here's our view object.

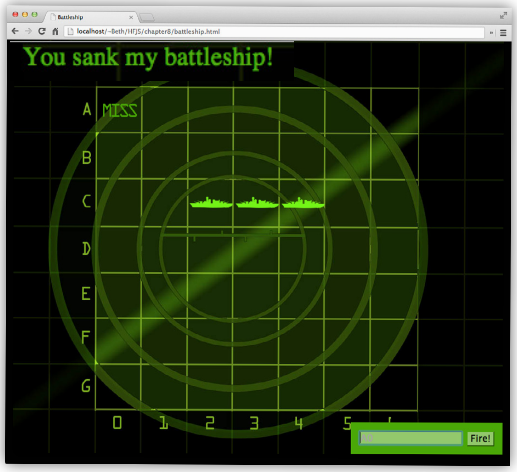
```
let view = {
    displayMessage(msg) {

    },
    displayHit(location) {
    },
    displayMiss(location) {
    }
};
```

We're going to start here.

How displayMessage works

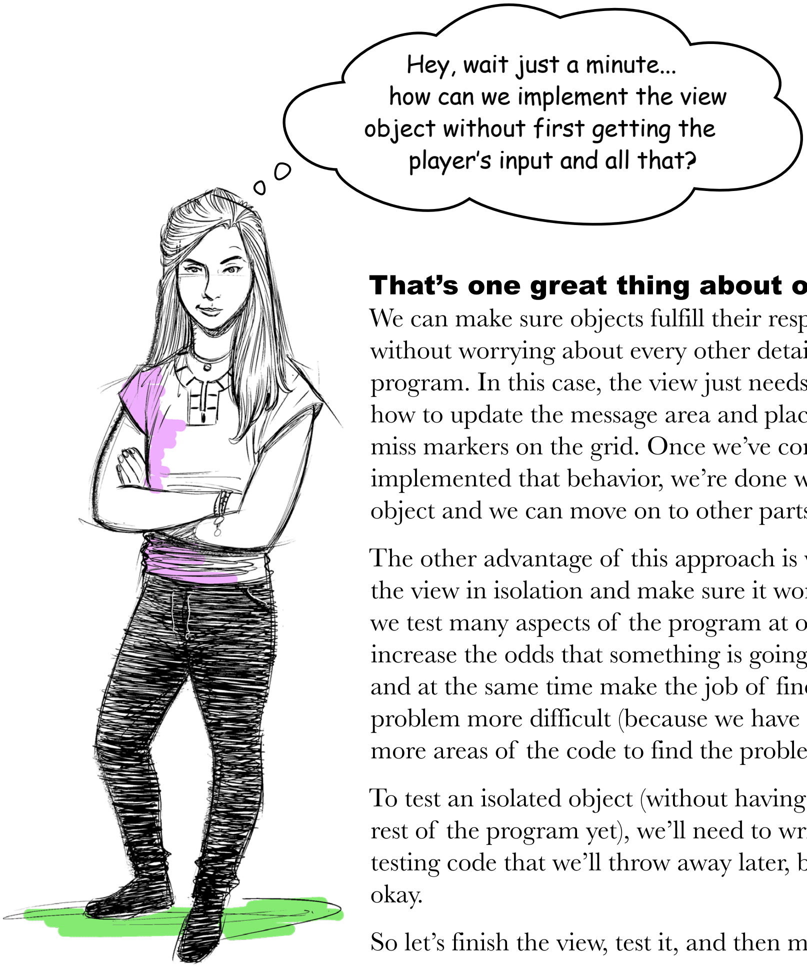
To implement the `displayMessage` method, you need to review the HTML and see that we have a `<div>` with the id "messageArea" ready for messages:

```
<div id="board">
  <div id="messageArea"></div>
    ...
</div>
```



You sank my battleship!

We'll use the DOM to get access to this `<div>`, and then set its text using `innerHTML`. And remember, whenever you change the DOM, you'll see the changes immediately in the browser. Here's what we're going to do…


Hey, wait just a minute... how can we implement the view object without first getting the player's input and all that?

That's one great thing about objects. We can make sure objects fulfill their responsibilities without worrying about every other detail of the program. In this case, the view just needs to know how to update the message area and place hit and miss markers on the grid. Once we've correctly implemented that behavior, we're done with the view object and we can move on to other parts of the code.

The other advantage of this approach is we can test the view in isolation and make sure it works. When we test many aspects of the program at once, we increase the odds that something is going to go wrong and at the same time make the job of finding the problem more difficult (because we have to examine more areas of the code to find the problem).

To test an isolated object (without having finished the rest of the program yet), we'll need to write a little testing code that we'll throw away later, but that's okay.

So let's finish the view, test it, and then move on!

Implementing displayMessage

Let's get back to writing the code for `displayMessage`. Remember, it needs to:

- Use the DOM to get the element with the id "messageArea".
- Set that element's `innerHTML` property to the message passed to the `displayMessage` method.

So, open up your blank “battleship.js” file, and add the view object:

```
let view = {
    displayMessage(msg) {
        let messageArea = document.getElementById("messageArea");
        messageArea.innerHTML = msg;
    },
    displayHit(location) {
    },
    displayMiss(location) {
    }
};
```

The displayMessage method takes one argument, a msg.

We get the messageArea element from the page...

...and update the text of the messageArea element by setting its innerHTML property to msg.

Before we test this code, let’s go ahead and write the other two methods. They won’t be incredibly complicated methods, and this way we can test the entire object at once.

How displayHit and displayMiss work

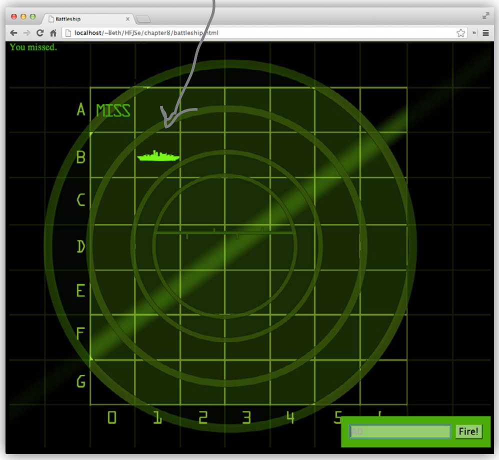
We just talked about this, but remember, to have an image appear on the game board, we need to take a `<td>` element and add either the “hit” or the “miss” class to the element. The former results in a “ship.png” appearing in the cell and the latter results in a “miss.png” being displayed.

We can affect the display by adding the “hit” or “miss” class to the `<td>` elements. Now we just need to do this from code.

```
<tr>
<td id="10"></td> <td class="hit" id="11"></td> <td id="12"></td> ...
</tr>
```

Now, in code, we’re going to use the DOM to get access to a `<td>`, and then set its class attribute to “hit” or “miss” using the `setAttribute` element method. As soon as we set the class attribute, you’ll see the appropriate image appear in the browser. Here’s what we’re going to do:

- Get a string id that consists of two numbers for the location of the hit or miss.
- Use the DOM to get the element with that id.
- Set that element’s class attribute to “hit” if we’re in `displayHit` or “miss” if we’re in `displayMiss`.


You missed.

Implementing displayHit and displayMiss

`displayHit` and `displayMiss` are methods that take the location of a hit or miss as an argument. That location should match the id of a cell (or `<td>` element) in the table representing the game board in the HTML. So, the first thing we need to do is get a reference to that element with the `getElementById` method. Let's try this in the `displayHit` method:

```
displayHit(location) {
    let cell = document.getElementById(location);
},
```

Remember, the location is created from the row and column and matches an id of a <td> element.

The next step is to add the class "hit" to the cell, which we can do with the `setAttribute` method like this:

```
displayHit(location) {
    let cell = document.getElementById(location);
    cell.setAttribute("class", "hit");
},
```

We then set the class of that element to "hit". This will immediately add a ship image to the <td> element.

Now let's add this code to the view object, and write `displayMiss` as well:

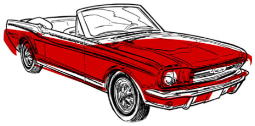
```
let view = {
    displayMessage(msg) {
        let messageArea = document.getElementById("messageArea");
        messageArea.innerHTML = msg;
    },

    displayHit(location) {
        let cell = document.getElementById(location);
        cell.setAttribute("class", "hit");
    },

    displayMiss(location) {
        let cell = document.getElementById(location);
        cell.setAttribute("class", "miss");
    }
};
```

We're using the id we created from the player's guess to get the correct element to update...

...and then setting the class of that element to "hit".

We do the same thing in displayMiss, only we set the class to "miss", which adds a MISS image to the element.

Make sure you add the code for `displayHit` and `displayMiss` to your "battleship.js" file.

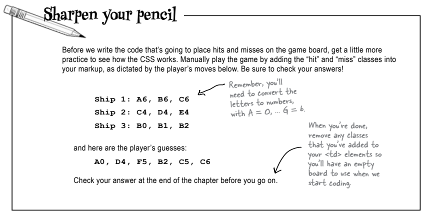


Another test drive...

Let's put the code through its paces before moving on…in fact, let's take the guesses from the previous Sharpen your Pencil exercise and implement them in code. Here's the sequence we want to implement:

```
A0, D4, F5, B2, C5, C6
```

MISS HIT MISS HIT MISS HIT



Sharpen your pencil

Before we write the code that's going to place hits and misses on the game board, get a little more practice to see how the CSS works. Manually play the game by adding the "hit" and "miss" classes into your markup, as dictated by the player's moves below. Be sure to check your answers!

```
Ship 1: A6, B6, C6
Ship 2: C4, D4, E4
Ship 3: B0, B1, B2
```

Remember, you'll need to convert the letters to numbers, with A = 0, … G = 6.

and here are the player's guesses:

```
A0, D4, F5, B2, C5, C6
```

Check your answer at the end of the chapter before you go on.

When you're done, remove any classes that you've added to your <td> elements so you'll have an empty board to use when we start coding.

To represent that sequence in code, add this to the bottom of your "battleship.js" JavaScript file:

```
view.displayMiss("00");
view.displayHit("34");
view.displayMiss("55");
view.displayHit("12");
view.displayMiss("25");
view.displayHit("26");
```

"A0"
"D4"
"F5"
"B2"
"C5"
"C6"

Remember, displayHit and displayMiss take a location on the board that's already been converted from a letter and a number to a string with two numbers that corresponds to the id of one of the table cells.

And let's not forget to test `displayMessage`:

```
view.displayMessage("Tap tap, is this thing on?");
```

Any message will do for simple testing...

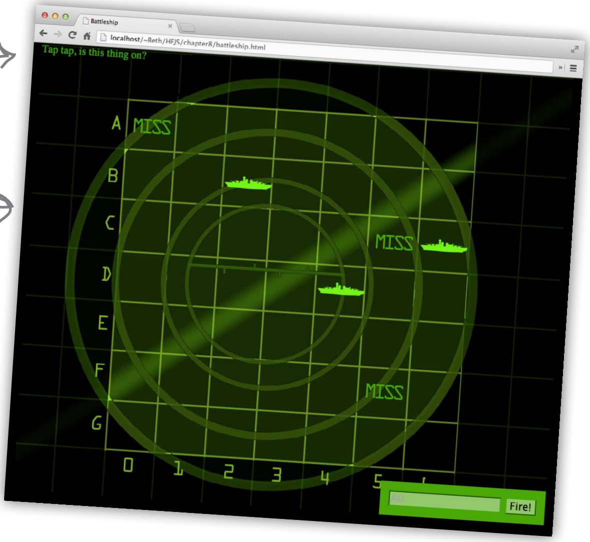
After all that, reload the page in your browser and check out the updates to the display.

One of the benefits of breaking up the code into objects and giving each object only one responsibility is that we can test each object to make sure it's doing its job correctly.

The "tap tap" message is displayed up here, at the top left of the view.

And the hits and misses we displayed using the view object are displayed on the game board.

Check each one to make sure it's in the right spot.

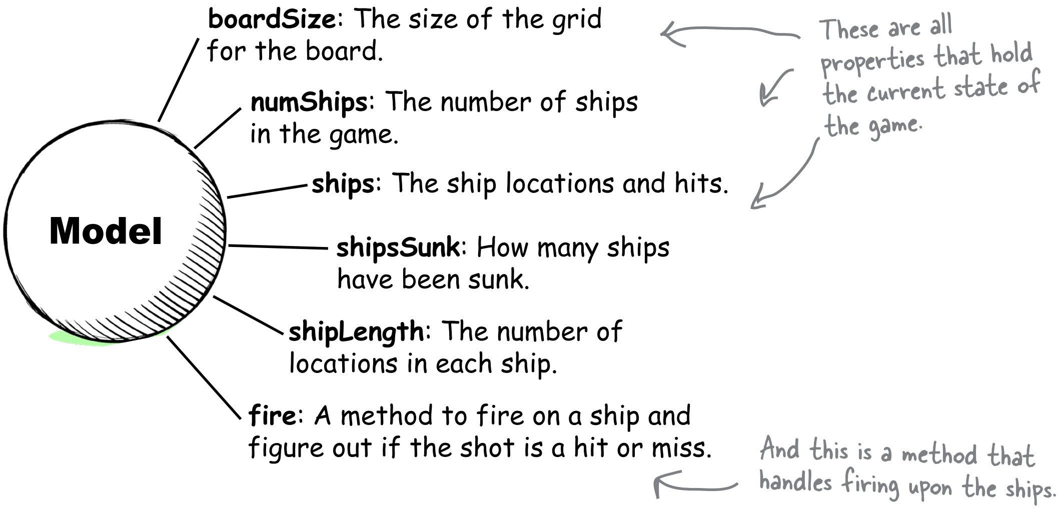


Tap tap, is this thing on?

The model

With the view object out of the way, let's move on to the model. The model is where we keep the *state* of the game. The model often also holds some *logic* relating to how the state changes. In this case, the state includes the locations of the ships, the ship locations that have been hit, and how many ships have been sunk. The only logic we're going to need (for now) is determining when a player's guess has hit a ship and then marking that location with a hit.

Here's what the model object is going to look like:

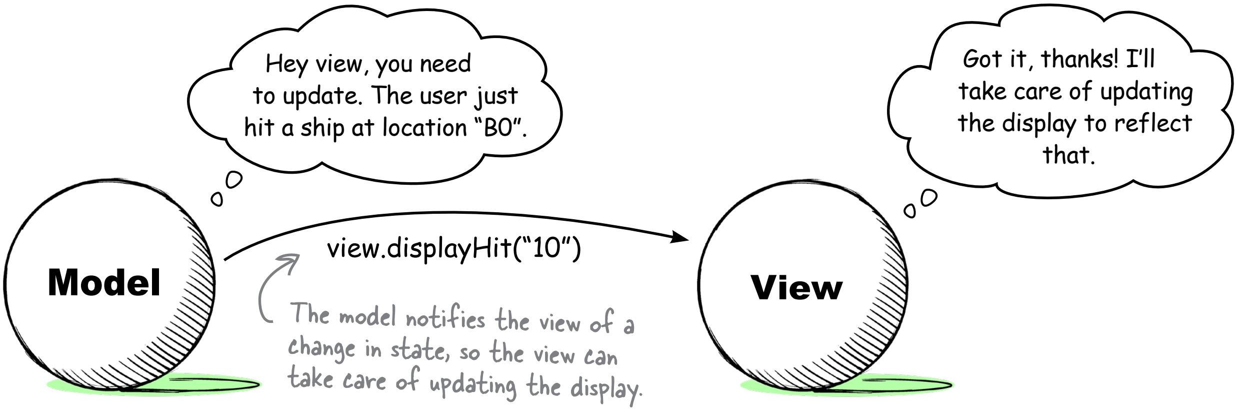

Model
boardSize: The size of the grid for the board.
numShips: The number of ships in the game.
ships: The ship locations and hits.
shipsSunk: How many ships have been sunk.
shipLength: The number of locations in each ship.
fire: A method to fire on a ship and figure out if the shot is a hit or miss.

These are all properties that hold the current state of the game.

And this is a method that handles firing upon the ships.

How the model interacts with the view

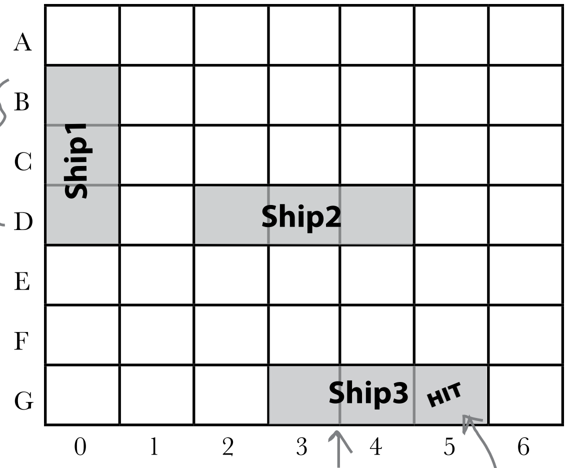
When the state of the game changes—that is, when you hit a ship, or miss—then the view needs to update the display. To do this, the model needs to talk to the view, and luckily we have a few methods the model can use to do that. We'll get our game logic set first in the model, then we'll add code to update the view.


Hey view, you need to update. The user just hit a ship at location "B0".
Got it, thanks! I'll take care of updating the display to reflect that.
Model
view.displayHit("10")
View

The model notifies the view of a change in state, so the view can take care of updating the display.

You're gonna need a bigger boat...and game board

Before we start writing model code, we need to think about how to represent the state of the ships in the model. Back in Chapter 2, in the Simple Battleship game, we had a single ship that sat on a 1 by 7 game board. Now things are a little more complex: we have *three* ships on a 7 by 7 board. Here's how it looks now:



Each ship takes up three cells on the 2D board.

This ship sits at locations B0, C0, D0.

Here's another ship at D2 through D4.

And a third ship at G3–G5.

You'll note that the ships don't overlap on the board. That would be impossible on a physical Battleship board and would lead to weird gameplay. We'll see how to make sure ships don't overlap later, when we talk about how to randomly place ships on the board.

We also need to be able to keep track of hits. Each ship has three locations, so we need to store three hits for each ship too.

Sharpen your pencil

Given how we've described the new game board above, how would you represent the ships in the model? (Just the locations; we'll worry about hits later.) Check off the best solution below.

☐ Use nine variables for the ship locations, similar to the way we handled the ships in Chapter 2.

☐ Use an array with an item for each cell in the board (49 items total). Record the ship number in each cell that holds part of a ship.

☐ Use an array to hold all nine locations. Items 0–2 will hold the first ship, 3–5 the second, and so on.

☐ Use three different arrays, one for each ship, with three locations contained in each.

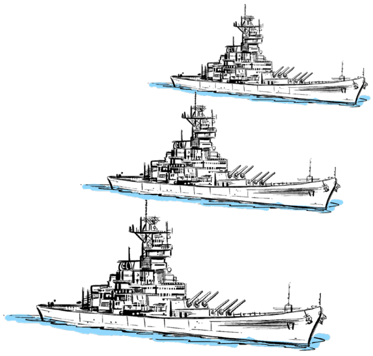
☐ Use an object named ship with three location properties. Put all the ships in an array named ships.

☐ ______________________________
____
____

Or write in your own answer.

Solution on page 392

How we're going to represent the ships



As you can see, there are many ways we can represent ships, and you may have even come up with a few other ways of your own. You'll find that no matter what kind of data you've got, there are many choices for storing that data, with various trade-offs depending on your choice—some methods will be space-efficient, others will optimize runtime, some will just be easier to understand, and so on.

We've chosen a representation for ships that is fairly simple—we're representing each ship as an object that holds the locations it sits in, along with the hits it's taken. Let's take a look at how we represent one ship:

Each ship is an object.

The ship has a locations property and a hits property.

```
let ship1 = {
              locations: ["10", "20", "30"],
              hits: ["", "", ""]
            };
```

The locations property is an array that holds each location on the board.

The hits property is also an array that holds whether or not a ship is hit at each location. We'll set the array items to the empty string initially, and change each item to "hit" when the ship has taken a hit in the corresponding location.

Note that we've converted the ship locations to two numbers, using 0 for A, 1 for B, and so on.

Here's what all three ships would look like:

Each ship has an array of three locations and an array to track hits.

```
let ship1 = { locations: ["10", "20", "30"], hits: ["", "", ""] };
let ship2 = { locations: ["32", "33", "34"], hits: ["", "", ""] };
let ship3 = { locations: ["63", "64", "65"], hits: ["", "", "hit"] };
```

And, rather than managing three different variables to hold the ships, we'll create a single array variable to hold them all, like this:

Note the plural name, ships.

We're assigning to ships an array that holds all three ships.

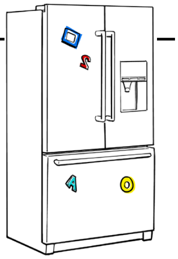
```
let ships = [{ locations: ["10", "20", "30"], hits: ["", "", ""] },
             { locations: ["32", "33", "34"], hits: ["", "", ""] },
             { locations: ["63", "64", "65"], hits: ["", "", "hit"] }];
```

Here's the first ship...

...and the second...

...and the third.

Note this ship has a hit at location "65" on the grid.



Ship Magnets

Use the following player moves, along with the data structure for the ships, to place the ship and MISS magnets onto the game board. Does the player sink all the ships? We've done the first move for you.

Here are the moves:

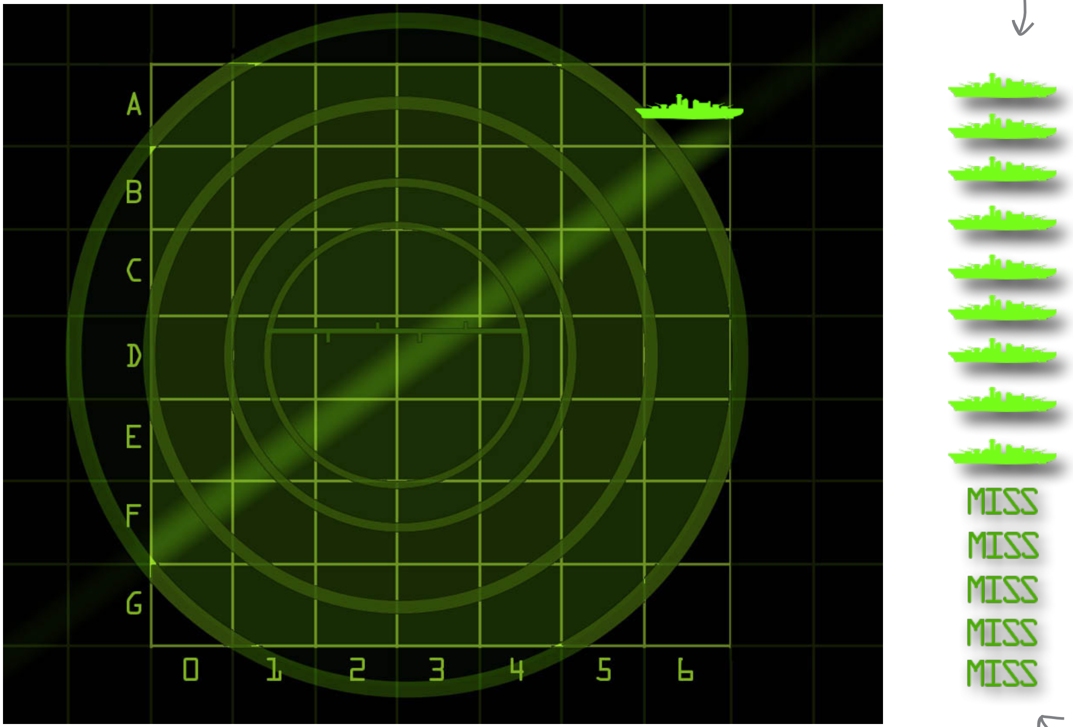
A6, B3, C4, D1, B0, D4, F0, A1, C6, B1, B2, E4, B6

Execute these moves on the game board.

```
let ships = [{ locations: ["06", "16", "26"], hits: ["hit", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] }];
```

Here is the data structure. Mark each ship with a hit as the game is played.

And here's the board and your magnets.



You might have leftover magnets.

Solution on page 393

Sharpen your pencil

Let's practice using the ships data structure to simulate some ship activities. Using the ships definition below, work through the following questions and code and fill in the blanks. Make sure you check your answers before moving on, as this is an important part of how the game works.

```
let ships = [{ locations: ["31", "41", "51"], hits: ["", "", ""] },
             { locations: ["14", "24", "34"], hits: ["", "hit", ""] },
             { locations: ["00", "01", "02"], hits: ["hit", "", ""] }];
```

Which ships are already hit? ____________ At what locations? ____________

The player guesses "D4"; does that hit a ship? ____________ If so, which one? ____________

The player guesses "B3"; does that hit a ship? ____________ If so, which one? ____________

Finish this code to access the second ship's middle location and print its value with console.log:

```
let ship2 = ships[____];
let locations = ship2.locations;
console.log("Location is " + locations[____]);
```

Finish this code to see if the third ship has a hit in its first location:

```
let ship3 = ships[____];
let hits = ship3._____;
if (_____ === "hit") {
   console.log("Ouch, hit on third ship at location one");
}
```

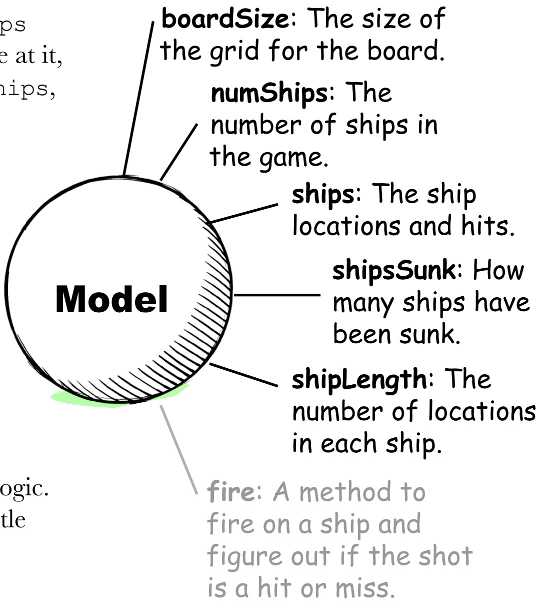
Finish this code to hit the first ship at the third location:

```
let ______ = ships[0];
let hits = ship1._______;
hits[____] = ________;
```

→ Solution on page 394

Implementing the model object

Now that you know how to represent the ships and the hits, let's get some code down. First we'll create the model object, then we'll take the `ships` data structure we just created and add it as a property. And while we're at it, there are a few other properties we're going to need as well, like `numShips`, to hold the number of ships we have in the game. You may be thinking, "What do you mean? We know there are three ships, why do we need a `numShips` property?" Well, what if you wanted to create a new version of the game that was more difficult and had four or five ships? By not "hardcoding" this value, and using a property instead (and then using the property throughout the code rather than the number), we can save ourselves a future headache if we need to change the number of ships, because we'll only need to change it in one place.



boardSize: The size of the grid for the board.

numShips: The number of ships in the game.

ships: The ship locations and hits.

shipsSunk: How many ships have been sunk.

shipLength: The number of locations in each ship.

fire: A method to fire on a ship and figure out if the shot is a hit or miss.

Now, speaking of hardcoding, we *are* going to hardcode the ships' initial locations, for now. Knowing where the ships are, will allow us to test the game more easily and focus on the core game logic. We'll tackle the code for placing random ships on the game board a little later.

So, let's get the model object created:

The model is an object.

```
let model = {
   boardSize: 7,
   numShips: 3,
   shipLength: 3,
   shipsSunk: 0,

   ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },
           { locations: ["24", "34", "44"], hits: ["", "", ""] },
           { locations: ["10", "11", "12"], hits: ["", "", ""] }]

};
```

These three properties keep us from hardcoding values. They are: boardSize (the size of the grid used for the board), numShips (the number of ships in the game), and shipLength (the number of locations in each ship).

shipsSunk (initialized to 0 for the start of the game) keeps the current number of ships that have been sunk by the player.

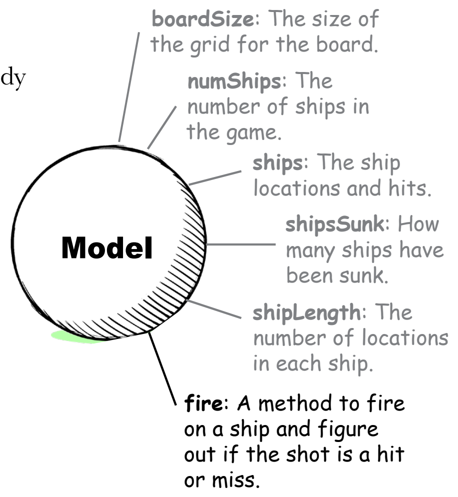
We've got quite a bit of state already!

Later on, we'll generate these locations for the ships so they're random, but for now, we'll hardcode them to make it easier to test the game.

The property ships is an array of ship objects that each store the locations and hits of one of the three ships. (Notice that we've changed ships from a variable, which we used before, to a property of the model object.)

Note we're also hardcoding the sizes of the locations and hits arrays. You'll learn how to dynamically create arrays later in the book.

Thinking about the fire method


Model
boardSize: The size of the grid for the board.
numShips: The number of ships in the game.
ships: The ship locations and hits.
shipsSunk: How many ships have been sunk.
shipLength: The number of locations in each ship.
fire: A method to fire on a ship and figure out if the shot is a hit or miss.

The `fire` method is what turns a player's guess into a hit or a miss. We already know the view object is going to take care of displaying the hits and misses, but the `fire` method has to provide the game logic for determining if a hit or a miss has occurred.

Knowing that a ship is hit is straightforward; given a player's guess, we just need to:

- Examine each ship and see if it occupies that location.
- If it does, we have a hit, and we'll mark the corresponding item in the `hits` array (and let the view know we got a hit). We'll also return `true` from the method, meaning we got a hit.
- If no ship occupies the guessed location, we've got a miss. We'll let the view know, and return `false` from the method.

The `fire` method should also determine if a ship isn't just hit, but sunk. We'll worry about that once we have the rest of the logic worked out.

Setting up the fire method

Let's get the basic skeleton of the `fire` method set up. The method will take a guess as an argument, and then iterate over each ship to determine if that ship was hit. We won't write the hit detection code just yet, but let's get the rest in place now:

```
let model = {
   boardSize: 7,
   numShips: 3,
   shipsSunk: 0,
   shipLength: 3,
   ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },
           { locations: ["24", "34", "44"], hits: ["", "", ""] },
           { locations: ["10", "11", "12"], hits: ["", "", ""] }],

   fire(guess) {

       for (let i = 0; i < this.numShips; i++) {
           let ship = this.ships[i];
       }
    }
};
```

Don't forget to add a comma here!

The method accepts a guess.

Then, we iterate through the array of ships, examining one ship at a time.

Here we have our hands on a ship. We need to see if the guess matches any of its locations.

Looking for hits

Now, each time through the loop, we need to see if the guess is one of the locations of the ship:

```
for (let i = 0; i < this.numShips; i++) {
    let ship = this.ships[i];
    let locations = ship.locations;

}
```

We're stepping through each ship.

And we've accessed the ship's set of locations. Remember, this is a property of the ship that contains an array.

What we need is the code that determines if the guess is in this ship's locations.

Here's the situation: we have a string, `guess`, that we're looking for in an array, `locations`. If `guess` matches one of those locations, we know we have a hit:

```
guess = "16";
locations = ["06", "16", "26"];
```

We need to find out if the value in guess is one of the values in the ship's locations array.

We could write yet another loop to go through each item in the `locations` array, compare the item to `guess`, and, if they match, in which case we have a hit.

But rather than write another loop, we have an easier way to do this:

```
let index = locations.indexOf(guess);
```

The indexOf method searches an array for a matching value and returns its index, or -1 if it can't find it.

So, using `indexOf`, we can write the code to find a hit like this:

```
for (let i = 0; i < this.numShips; i++) {
    let ship = this.ships[i];
    let locations = ship.locations;
    let index = locations.indexOf(guess);
    if (index >= 0) {
        // We have a hit!
    }
}
```

Notice that the indexOf method for an array is similar to the indexOf string method. It takes a value and returns the index of that value in the array (or -1 if it can't find the value).

So if we get an index greater than or equal to zero, the user's guess is in the location's array, and we have a hit.

Using `indexOf` isn't any more efficient than writing a loop, but it is a little clearer and it's definitely less code. We'd also argue that the intent of this code is clearer than if we wrote a loop: it's easier to see what value we're looking for in an array using `indexOf`. In any case, you now have another tool in your programming toolbelt.

Putting it all together

To finish this up, we have one more thing to determine: if we have a hit, what do we do? All we need to do for now is mark the hit in the model, which means adding a "hit" string to the `hits` array. Let's put all the pieces together:

```
let model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],

     fire(guess) {
         for (let i = 0; i < this.numShips; i++) {
             let ship = this.ships[i];
             let locations = ship.locations;
             let index = locations.indexOf(guess);
             if (index >= 0) {
                 ship.hits[index] = "hit";
                 return true;
             }
         }
         return false;
     }
};
```

For each ship...

...if the guess is in the locations array, we have a hit.

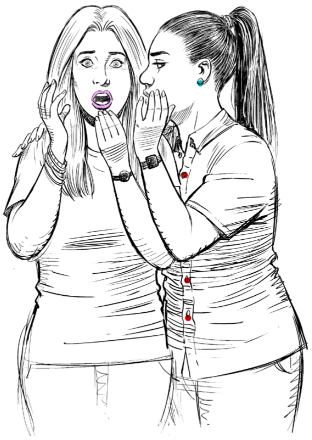
So we mark the hits array at the same index.

Oh, and we need to return true because we had a hit.

Otherwise, if we make it through all the ships and don't have a hit, it's a miss, so we return false.

That's a great start on our model object. There are only a couple of other things we need to do: determine if a ship is sunk, and let the view know about the changes in the model so it can keep the player updated. Let's get started on those…

Wait, can we talk about your verbosity again?



Sorry, we have to bring this up again. You're being a bit verbose in some of your references to objects and arrays. Take another look at the code:

```
for (let i = 0; i < this.numShips; i++) {
    let ship = this.ships[i];
    let locations = ship.locations;
    let index = locations.indexOf(guess);
    ...
}
```

First we get the ship.

Then we get the locations in the ship.

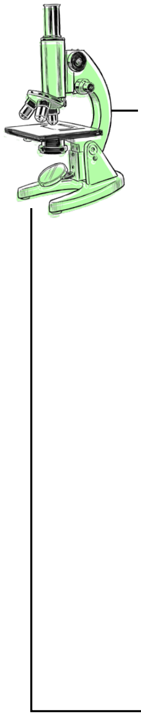
Then we get the index of the guess in the locations.

Some would call this code overly verbose. Why? Because some of these references can be shortened using *chaining*. Chaining allows us to string together object references so that we don't have to create temporary variables, like the `locations` variable in the code above.

Now, you might ask why `locations` is a temporary variable. That's because we're using `locations` only to temporarily store the `ship.locations` array, so we can then turn around and call the `indexOf` method on it to get the index of the `guess`. We don't need `locations` for anything else in this method. With chaining, we can get rid of that temporary `locations` variable, like this:

```
let index = ship.locations.indexOf(guess);
```

We've combined the two lines highlighted above into a single line.



How chaining works...

Chaining is really just a shorthand for a longer series of steps to access properties and methods of objects (and arrays). Let's take a closer look at what we just did to combine two statements with chaining:

Here's a ship object.

```
let ship = { locations: ["06", "16", "26"], hits: ["", "", ""] };
let locations = ship.locations;
let index = locations.indexOf(guess);
```

We were grabbing the locations array from the ship...

...then using it to access the indexOf method.

We can combine the bottom two statements by chaining together the expressions (and getting rid of the variable `locations`):

```
ship.locations.indexOf(guess)
```

1. Evaluates to the ship object....
2. ...which has a locations property, which is an array...
3. ...which has a method named indexOf.

Meanwhile, back at the battleship...

Now we need to write the code to determine if a ship is sunk. You know the rules: a battleship is sunk when all of its locations are hit. We can add a little helper method to check to see if a ship is sunk:

We'll call the method isSunk. It's going to take a ship and return true if it's sunk and false if it is still floating.

This method takes a ship, and then checks to see if there are any locations that don't yet have a hit.

```
isSunk(ship) {
    return (!ship.hits.includes(""));
}
```

If there's a location that doesn't have a hit, then the ship is still floating, so return false; otherwise, return true.

Sharpen your pencil

To see how isSunk works, work through these examples. We've done one for you. Hint: includes returns true if the array contains the argument anywhere in the array; otherwise, it returns false (just like with strings).

`ship.hits`	`ship.hits.includes("")`	`!ship.hits.includes("")`
`["", "", ""]`	true	false
`["", "hit", ""]`	________	________
`["hit", "hit", "hit"]`	________	________

→ Solution on page 395

Now, we can use the `isSunk` method in the `fire` method to find out if a ship is sunk:

```
fire(guess) {
    for (let i = 0; i < this.numShips; i++) {
        let ship = this.ships[i];
        let index = ship.locations.indexOf(guess);
        if (index >= 0) {
            ship.hits[index] = "hit";
            if (this.isSunk(ship)) {
                this.shipsSunk++;
            }
            return true;
        }
    }
    return false;
},
isSunk(ship) { ... }
```

We'll add the check here, after we know for sure we have a hit. If the ship is sunk, then we increase the number of ships that are sunk in the model's shipsSunk property.

Here's where we added the new isSunk method, just below fire. Don't forget to make sure you've got a comma between each of the model's properties and methods!

A view to a kill...

That's about it for the model object. The model maintains the state of the game and has the logic to test guesses for hits and misses. The only thing we're missing is the code to notify the view when we get a hit or a miss in the model. Let's add that now:

```
let model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],
    fire(guess) {
        for (let i = 0; i < this.numShips; i++) {
            let ship = this.ships[i];
            let index = ship.locations.indexOf(guess);
            if (index >= 0) {
                ship.hits[index] = "hit";
                view.displayHit(guess);
                view.displayMessage("HIT!");
                if (this.isSunk(ship)) {
                    view.displayMessage("You sank my battleship!");
                    this.shipsSunk++;
                }
                return true;
            }
        }
        view.displayMiss(guess);
        view.displayMessage("You missed.");
        return false;
    },
    isSunk(ship) {
        return (!ship.hits.includes(""));
    }
};
```

This is the whole model object, so you can see the entire thing in one piece.

Notify the view that we got a hit at the location in guess.

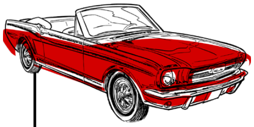
And ask the view to display the message "HIT!"

Let the player know that this hit sank the battleship!

Notify the view that we got a miss at the location in guess.

And ask the view to display the message "You missed."

Remember that the methods in the view object add the "hit" or "miss" class to the element with the id at the row and column in the guess string, so the view translates the "hit" in the hits array into a "hit" in the HTML. But keep in mind, the "hit" in the HTML is just for display; the "hit" in the model represents the actual state.

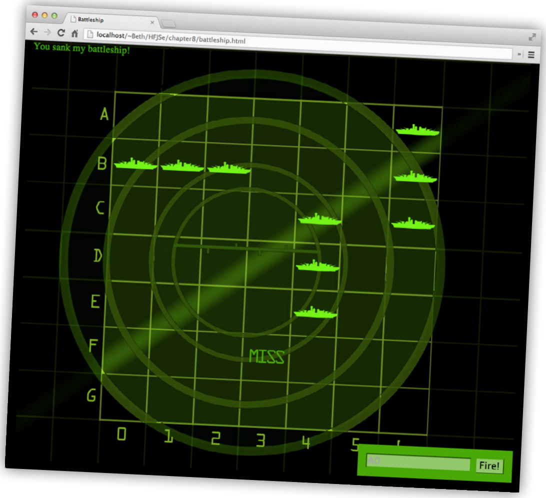


Test Drive

Add all the model code to “battleship.js”. Test it by calling the model’s fire method, passing in the row and column of a guess each time. Our ships are still hardcoded, so it’ll be easy for you to hit them all. Try adding a few more random guesses as well. (Download “battleship_tester.js” to see our version of the test code.)

You’ll need to remove or comment out the previous view testing code to get the same results as we show here. You can see how to do that in “battleship_tester.js”.

```
model.fire("53");

model.fire("06");
model.fire("16");
model.fire("26");

model.fire("34");
model.fire("24");
model.fire("44");

model.fire("12");
model.fire("11");
model.fire("10");
```



You sank my battleship!

Reload “battleship.html”. You should see your hits and misses appear on the game board.

there are no Dumb Questions

Q: Is using chaining to combine statements better than keeping statements separate?

A: Not necessarily better, no. Chaining isn’t much more efficient (you save one variable), but it does make your code shorter. We’d argue that short chains (two or three levels at most) are easier to read than multiple lines of code, but that’s our preference. If you want to keep your statements separate, that’s fine. And if you do use chaining, make sure you don’t create really long chains; they will be harder to read and understand if they’re too long.

Q: We have arrays (locations) inside an object (ship) inside an array (ships). How many levels deep can you nest objects and arrays like this?

A: Pretty much as deep as you want. Practically, of course, it’s unlikely you’ll ever go too deep (and if you find yourself with more than three or four levels of nesting, it’s likely your data structure is getting too complex and you should rethink things a bit).

Q: I noticed we added a property named boardSize to the model, but we haven’t used it in the model code. What is that for?

A: We’re going to be using model.boardSize, and the other properties in model, in the code coming up. The model’s responsibility is to manage the state of the game, and boardSize is definitely part of the state. The controller will access the state it needs by accessing the model’s properties, and we’ll be adding more model methods later that will use these properties too.

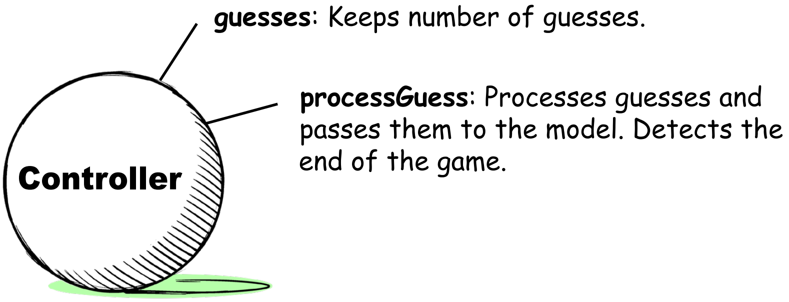
Implementing the controller

Now that we have the view and the model complete, we're going to start to bring this app together by implementing the controller. At a high level, the controller glues everything together by getting a guess, processing the guess, and getting it to the model. It also keeps track of some administrative details, like the current number of guesses and the player's progress in the game. To do all this the controller relies on the model to keep the state of the game and on the view to display the game.

More specifically, here's the set of responsibilities we're giving the controller:

- Get and process the player's guess (like "A0" or "B1").
- Keep track of the number of guesses.
- Ask the model to update itself based on the latest guess.
- Determine when the game is over (that is, when all ships have been sunk).

Let's get started on the controller by first defining a property, `guesses`, in the controller object. Then we'll implement a single method, `processGuess`, that takes an alphanumeric guess, processes it, and passes it to the model.


guesses: Keeps number of guesses.
processGuess: Processes guesses and passes them to the model. Detects the end of the game.
Controller

Here's the skeleton of the controller code; we'll fill this in over the next few pages:

```
let controller = {
   guesses: 0,

   processGuess(guess) {
     // more code will go here
   }
};
```

Here we're defining our controller object, with a property, guesses, initialized to zero.

And here's the beginning of the processGuess method, which takes a guess in the form "A0".

Processing the player's guess

The controller's responsibility is to get the player's guess, make sure it's valid, and then get it to the model object. But where does it get the player's guess from? Don't worry, we'll get to that in a bit. For now, we're just going to assume that at some point some code is going to call the controller's `processGuess` method and give it a string in the form:

This is a great technique when you are coding. Focus on the requirements for the specific code you're working on. Thinking about the whole problem at once is often a less successful technique.

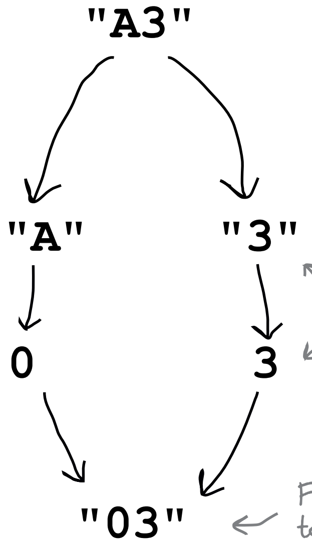
"A3"

You know the Battleship-style guess format at this point: it's a letter followed by a number.

After we receive a guess in this form (an alphanumeric set of characters, like "A3"), we'll need to transform the guess into a form the model understands (a string of two numeric characters, like "03"). Here's a high-level view of how we're going to convert a valid input into the number-only form:

Surely a player would never enter in an invalid guess, right? Ha! We'd better make sure we've got valid input.

Assume we've been handed a string in alphanumeric form:



First, we'll break the string apart, into the row and the column.

We'll convert the letter to a number and check to make sure it is between 0 and 6.

And we'll convert this to a number and check to make sure it is between 0 and 6.

Finally, we'll put the two numbers together back into a string.

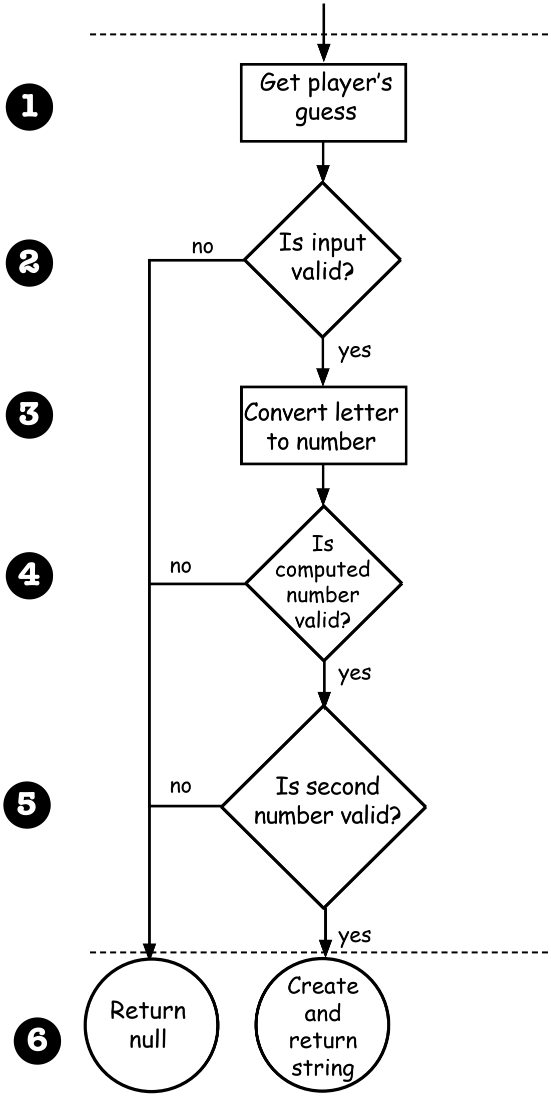
But first things first. We also need to check that the input is valid. Let's plan this all out before we write the code.

Planning the code...

Rather than putting all this guess-processing code into the `processGuess` method, we're going to write a little helper function (after all, we might be able to use this again). We'll name the function `parseGuess`.

Let's step through how it is going to work before we start writing code:

1. Get a player's guess in classic Battleship style, as a single letter followed by a number.
2. Check the input to make sure it is valid (not null or too long or too short).
3. Take the letter and convert it to a number: A to 0, B to 1, and so on.
4. See if the number from step 3 is valid (between 0 and 6).
5. Check the second number for validity (also between 0 and 6).
6. If any check failed, return null. Otherwise, concatenate the two numbers into a string and return the string.



1
Get player's guess
2
Is input valid?
no
yes
3
Convert letter to number
4
Is computed number valid?
no
yes
5
Is second number valid?
no
yes
6
Return null
Create and return string

Implementing parseGuess

We have a solid plan for coding this, so let's get started:

1—2 For steps 1 and 2, all we need to do is accept the player's guess and check to make sure it is valid. At this point we're just going to define validity as accepting a string that has exactly two characters in it:

The guess is passed into the guess parameter.

And then we check for null and to make sure the length is 2 characters.

```
function parseGuess(guess) {
   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   }
}
```

If not, we alert the player.

3 Next, we take the letter and convert it to a number by using a helper array that contains the letters A-G. To get the number, we can use the indexOf method to get the index of the letter in the array, like this:

An array loaded with each letter that could be part of a valid guess.

```
function parseGuess(guess) {
  const alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      let firstChar = guess.charAt(0);
      let row = alphabet.indexOf(firstChar);
   }
}
```

We grab the first character of the guess.

Then, using indexOf, we get back a number between 0 and 6 that corresponds to the letter, or -1 if firstChar doesn't match any of the letters in the array. Try a couple of examples to see how this works.

4—5 Now we'll handle checking both characters of the guess to see if they are numbers between 0 and 6 (in other words, to make sure they are both valid positions on the board):

```
function parseGuess(guess) {
   const alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      let firstChar = guess.charAt(0);
      let row = alphabet.indexOf(firstChar);
      let column = guess.charAt(1);

      if (isNaN(column)) {
         alert("Oops, that isn't on the board.");
      } else if (row < 0 || row >= model.boardSize ||
                 column < 0 || column >= model.boardSize) {
         alert("Oops, that's off the board!");
      }
   }
}
```

Here we've added code to grab the second character in the string, which represents the column.

And we're checking to see if column is not a number using the isNaN function. Remember your type conversion rules; to test isNaN, JavaScript will try to convert column to a number from a string, and then test.

Notice we're using type conversion like crazy here! column is a string, so when we check to make sure its value is 0–6, we rely on type conversion to convert it to a number for comparison.

Actually, we're being even more general here. Instead of hardcoding the number 6, we're asking the model to tell us how big the board is and using that number for comparison.

We're also making sure that the numbers are between 0 and 6.

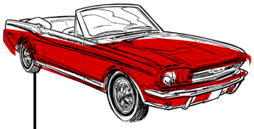
6 Now for our final bit of code for the parseGuess function...If any check for valid input fails, we'll return null. Otherwise, we'll return the row and column of the guess, combined into a string:

```
function parseGuess(guess) {
   const alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      let firstChar = guess.charAt(0);
      let row = alphabet.indexOf(firstChar);
      let column = guess.charAt(1);

      if (isNaN(column)) {
         alert("Oops, that isn't on the board.");
      } else if (row < 0 || row >= model.boardSize ||
                 column < 0 || column >= model.boardSize) {
         alert("Oops, that's off the board!");
      } else {
         return row + column;
      }
   }
   return null;
}
```

At this point, everything looks good, so we can return a row and column.

Notice we're concatenating the row and column together to make a string, and returning that string. We're using type conversion again here: row is a number and column is a string, so we'll end up with a string.

If we get here, there was a failed check along the way, so we return null.

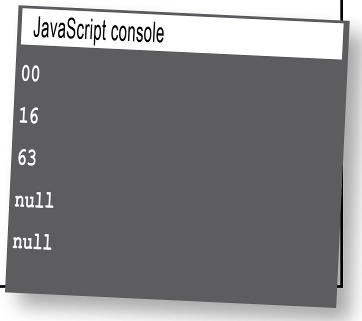


Test Drive

Make sure you've entered all this code into "battleship.js", and then add some function calls below it that look like this:

```
console.log(parseGuess("A0"));
console.log(parseGuess("B6"));
console.log(parseGuess("G3"));
console.log(parseGuess("H0"));
console.log(parseGuess("A7"));
```

Reload "battleship.html", and make sure your console window is open. You should see the results of parseGuess displayed in the console, and possibly an alert or two.



```
00
16
63
null
null
```

Meanwhile, back at the controller...

Now that we have the `parseGuess` helper function written, we can move on to implementing the controller. Let's first integrate the `parseGuess` function with the existing controller code:

```
let controller = {
   guesses: 0,

   processGuess(guess) {
      let location = parseGuess(guess);
      if (location) {

      }
   }
};
```

We'll use parseGuess to validate the player's guess.

As long as we don't get null back, we know we've got a valid location object.

Remember, null is a falsey value.

The rest of the code for the controller will go here.

That completes the first responsibility of the controller. Let's see what's left:

- ~~Get and process the player's guess (like "A0" or "B1").~~
- Keep track of the number of guesses.
- Ask the model to update itself based on the latest guess.
- Determine when the game is over (that is, when all ships have been sunk).

We'll tackle these next.

Counting guesses and firing the shot

The next item on our list is straightforward: to keep track of the number of guesses, we just need to increment the `guesses` property each time the player makes a guess. As you'll see in the code, we've chosen not to penalize players if they enter an invalid guess.

Next, we'll ask the model to update itself based on the guess by calling the model's `fire` method. After all, the point of a player's guess is to fire hoping to hit a battleship. Now remember, the `fire` method takes a string, which contains the row and column, and by some luck we get that string by calling `parseGuess`. How convenient.

Let's put all this together and implement the next step…

```
let controller = {
    guesses: 0,

    processGuess(guess) {
        let location = parseGuess(guess);
        if (location) {
            this.guesses++;
            let hit = model.fire(location);
        }
    }
};
```

If the player entered a valid guess, we increase the number of guesses by one.

Remember, this.guesses++ just adds one to the value of the guesses property. It works just like i++ in for loops.

Also notice if the player enters an invalid board location, we don't penalize them by counting the guess.

And then we pass the row and column in the form of a string to the model's fire method. Remember, the fire method returns true if a ship is hit.

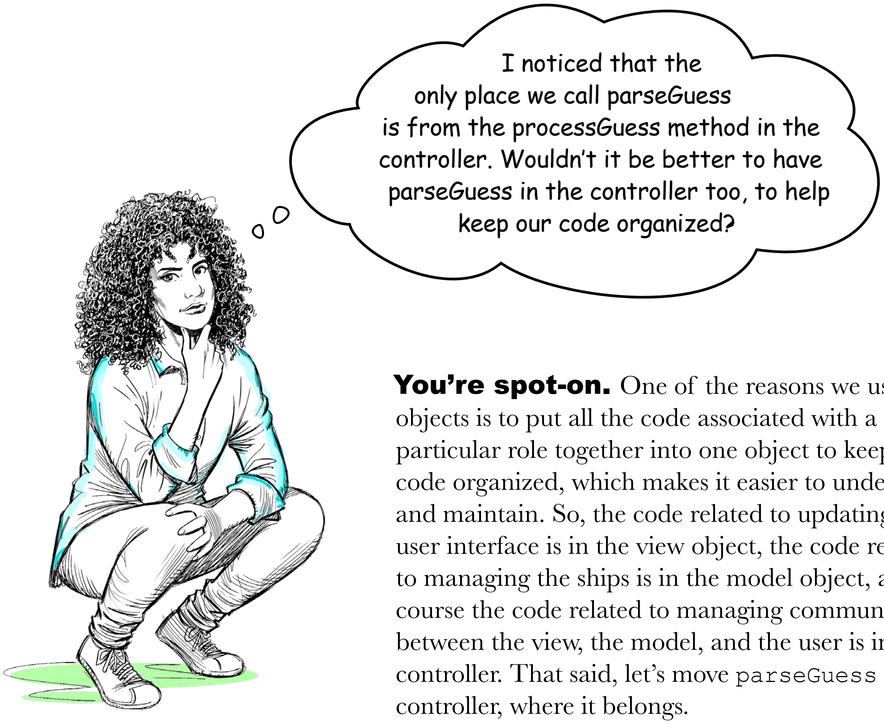
Game over?

All we have left is to determine when the game is complete. How do we do that? Well, we know that when three ships are sunk, the game is over. So, each time the guess is a hit, we'll check to see if there are three sunken ships, using the `model.shipsSunk` property. Let's generalize this a bit; instead of just comparing it to the number 3, we'll use the model's `numShips` property for the comparison. You might decide later to set the number of ships to, say, 2 or 4, and this way, you won't need to revisit this code to make it work correctly:

```
let controller = {
    guesses: 0,

    processGuess(guess) {
        let location = parseGuess(guess);
        if (location) {
            this.guesses++;
            let hit = model.fire(location);
            if (hit && model.shipsSunk === model.numShips) {
              view.displayMessage(`You sank all my battleships, in ${this.guesses} guesses`);
            }
        }
    }
};
```

If the guess was a hit, and the number of ships that are sunk is equal to the number of ships in the game, then show the player a message that they've sunk all the ships.

We'll show the player the total number of guesses they took to sink the ships. The guesses property is a property of **this** object, the controller.


I noticed that the only place we call parseGuess is from the processGuess method in the controller. Wouldn't it be better to have parseGuess in the controller too, to help keep our code organized?

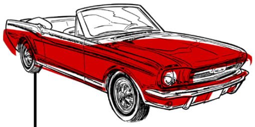
You're spot-on. One of the reasons we use objects is to put all the code associated with a particular role together into one object to keep our code organized, which makes it easier to understand and maintain. So, the code related to updating the user interface is in the view object, the code related to managing the ships is in the model object, and of course the code related to managing communication between the view, the model, and the user is in the controller. That said, let's move `parseGuess` into the controller, where it belongs.

Exercise

Here's what the code looks like now. Go ahead and move parseGuess into the controller object, before we do a Test Drive on the next page. Check our solution at the end of the chapter before you go on.

```
let controller = {
   guesses: 0,

   processGuess(guess) {
      let location = parseGuess(guess);
      // ... same as on previous page
   }
};

function parseGuess(guess) {
   const alphabet = ["A", "B", "C", "D", "E", "F", "G"];
   // rest of the code here (from a few pages back)
}
```

Solution on page 396

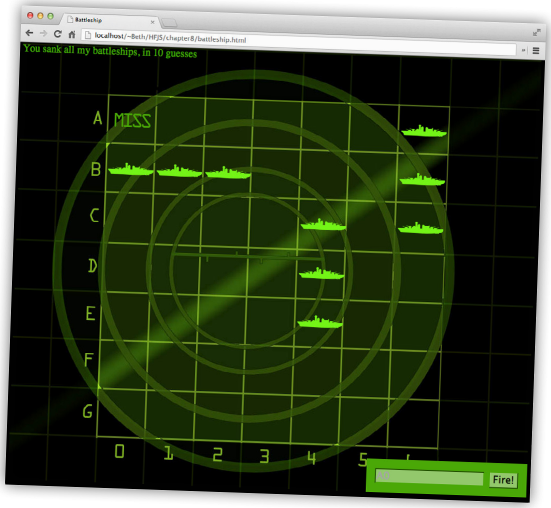


Test Drive

Okay, make sure you've entered all the controller code into your "battleship.js" file, and then add some function calls below it to test your controller. Reload your "battleship.html" page and note the hits and misses on the board. Are they in the right places? (Download "battleship_tester.js" to see our version.)

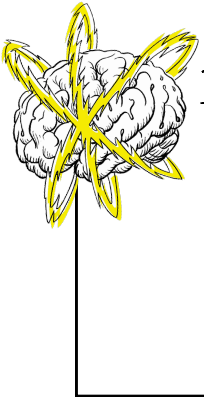
Again, you'll need to remove or comment out the previous testing code to get the same results as we show here. You can see how to do that in "battleship_tester.js".

```
controller.processGuess("A0");

controller.processGuess("A6");
controller.processGuess("B6");
controller.processGuess("C6");

controller.processGuess("C4");
controller.processGuess("D4");
controller.processGuess("E4");

controller.processGuess("B0");
controller.processGuess("B1");
controller.processGuess("B2");
```



You sank all my battleships, in 10 guesses

We're calling the controller's processGuess method and passing in guesses in Battleship format.

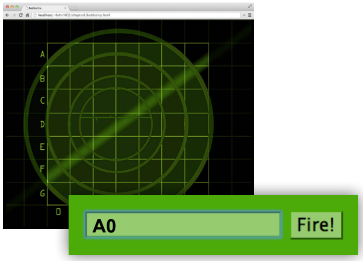


Brain Power

We let the player know the game ended in the message area, after they sink all three ships. But the player can still enter guesses. If you wanted to fix this so a player isn't allowed to enter guesses after they've sunk all the ships, how would you handle that?

Getting a player's guess

Now that you've implemented the core game logic and display, you need a way to let a player enter guesses and retrieve them so the game can actually be played. You might remember that in the HTML we've already got a `<form>` element ready for entering guesses, but how do we hook that into the game?



Here's our HTML `<form>` element, ready to take user input.

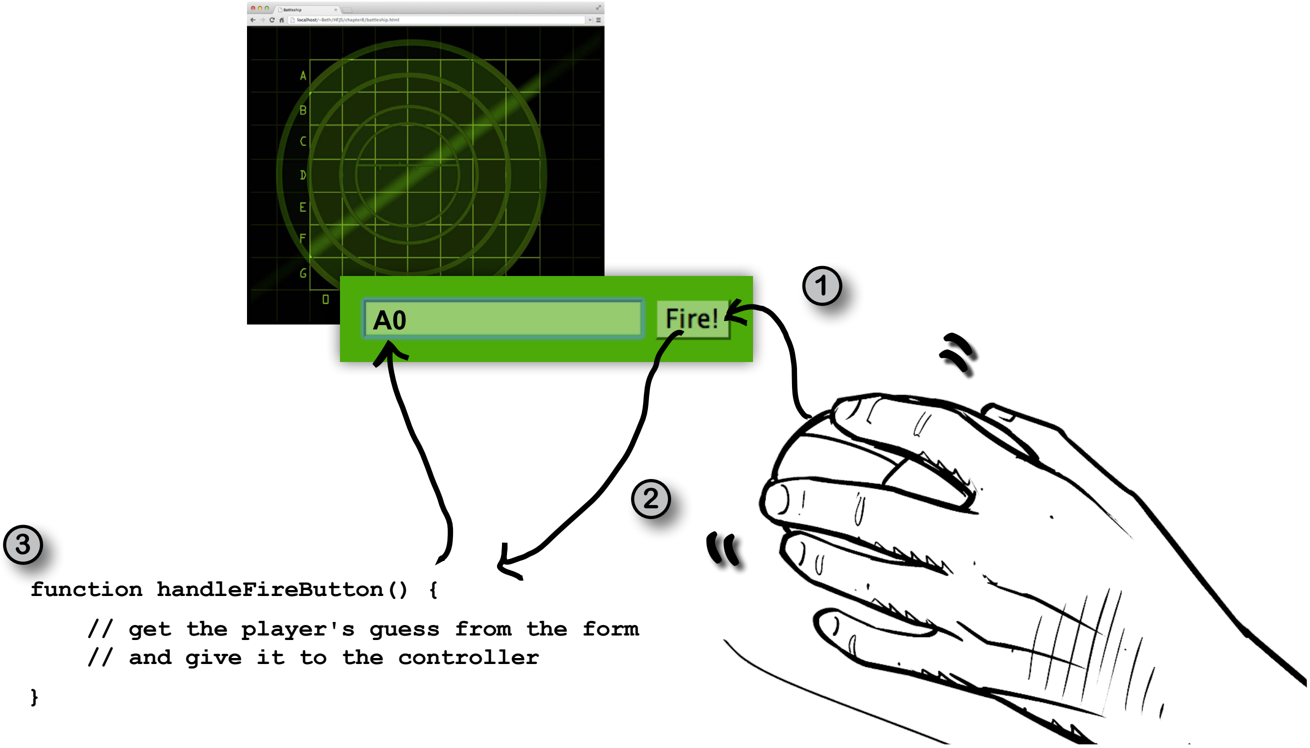
To do that, we need an *event handler*. We talked a little about event handlers Chapter 6. Here, we're going to spend just enough time with them to get the game working. Our goal is for you to get a high-level understanding of how event handlers work with `<form>` elements; we'll get into all the nitty-gritty details in the next chapter.

Here's the big picture:

① The player enters a guess and clicks the Fire! button.

② When Fire! is clicked, a preassigned event handler is called.

③ The handler for the Fire! button grabs the player's input from the form and hands it to the controller.



```
function handleFireButton() {
    // get the player's guess from the form
    // and give it to the controller
}
```

Adding an event handler to the Fire! button

To get this all rolling, the first thing we need to do is add an event handler to the Fire! button. To do that, we first need to get a reference to the button using the button's id. Review your HTML again, and you'll find the Fire! button has the id "fireButton". With that, all we need to do is call `document.getElementById` to get a reference to the button. Once we have the button reference, we can assign a handler function to the `onclick` property of the button, like this:

We need somewhere for this code to go, so let's create an init function.

First, we get a reference to the Fire! button using the button's id.

```
function init() {
    let fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
}
```

Then we can add a click handler function named handleFireButton to the button.

And let's not forget to get a handleFireButton function started.

This function will be called whenever you click the Fire! button.

```
function handleFireButton() {
    // code to get the value from the form
}
```

We'll write this code in just a sec.

```
window.onload = init;
```

Just like you learned in Chapter 6, we want the browser to run init when the page is fully loaded.

Getting the player's guess from the form

The Fire! button is what initiates the guess, but the player's guess is actually contained in the "guessInput" form element. We can get the value from the form input by accessing the input element's `value` property, like this:

First, we get a reference to the form input element using the input element's id, "guessInput".

```
function handleFireButton() {
    let guessInput = document.getElementById("guessInput");
    let guess = guessInput.value;
}
```

Then we get the guess from the input element. The guess is stored in the value property of the input element.

We have the value; now we just need to do something with it. Luckily, we have lots of code already that's ready to put it to use. Let's add that next.

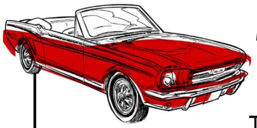
Passing the input to the controller

Here's where it all comes together. We have a controller waiting—just dying—to get a guess from the player. All we need to do is pass the player's guess to the controller. Let's do that:

```
function handleFireButton() {
    let guessInput = document.getElementById("guessInput");
    let guess = guessInput.value;
    controller.processGuess(guess);

    guessInput.value = "";
}
```

We're passing the player's guess to the controller, and then everything should work like magic!

This little line just resets the form input element to be the empty string. That way you don't have to explicitly select the text and delete it before entering the next guess, which would be annoying.



Test Drive

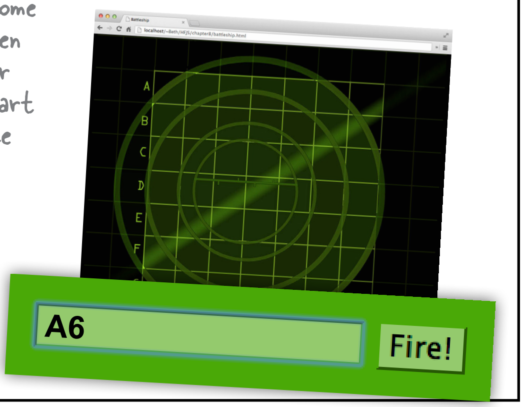
This is no mere test drive. You're finally ready to play the real game! Make sure you've added all the code to "battleship.js", and reload "battleship.html" in your browser. Now, remember the ship locations are hardcoded, so you'll have a good idea of how to win this game. Below you'll find the winning moves, but be sure to fully test the code. Enter misses, invalid guesses, and downright incorrect guesses.

A6
B6
C6

C4
D4
E4

B0
B1
B2

These are the winning guesses, in order by ship. But you don't have to enter them all in order. Try mixing them up a bit. Enter some invalid guesses in between the correct ones. Enter misses too. That's all part of the quality assurance testing for the game.



Geek Note

Finding it clumsy to have to click the Fire! button with every guess? Sure, clicking works, but it's slow and inconvenient. It would be so much easier if you could just press Return, right? Here's a quick bit of code to handle a Return key press:

```
function init() {
    let fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
    let guessInput = document.getElementById("guessInput");
    guessInput.onkeypress = handleKeyPress;
}
```

Add a new handler. This one handles key press events from the HTML input field.

Here's the key press handler. It's called whenever you press a key in the form input in the page.

The browser passes an event object to the handler. This object has info about which key was pressed.

```
function handleKeyPress(e) {
    let fireButton = document.getElementById("fireButton");
    if (e.keyCode === 13) {
        fireButton.click();
        return false;
    }
}
```

If you press the Return key, the event's keyCode property will be set to 13. If that's the case, then we want to cause the Fire! button to act like it was clicked. We can do that by calling the fireButton's click method (basically tricking it into thinking it was clicked).

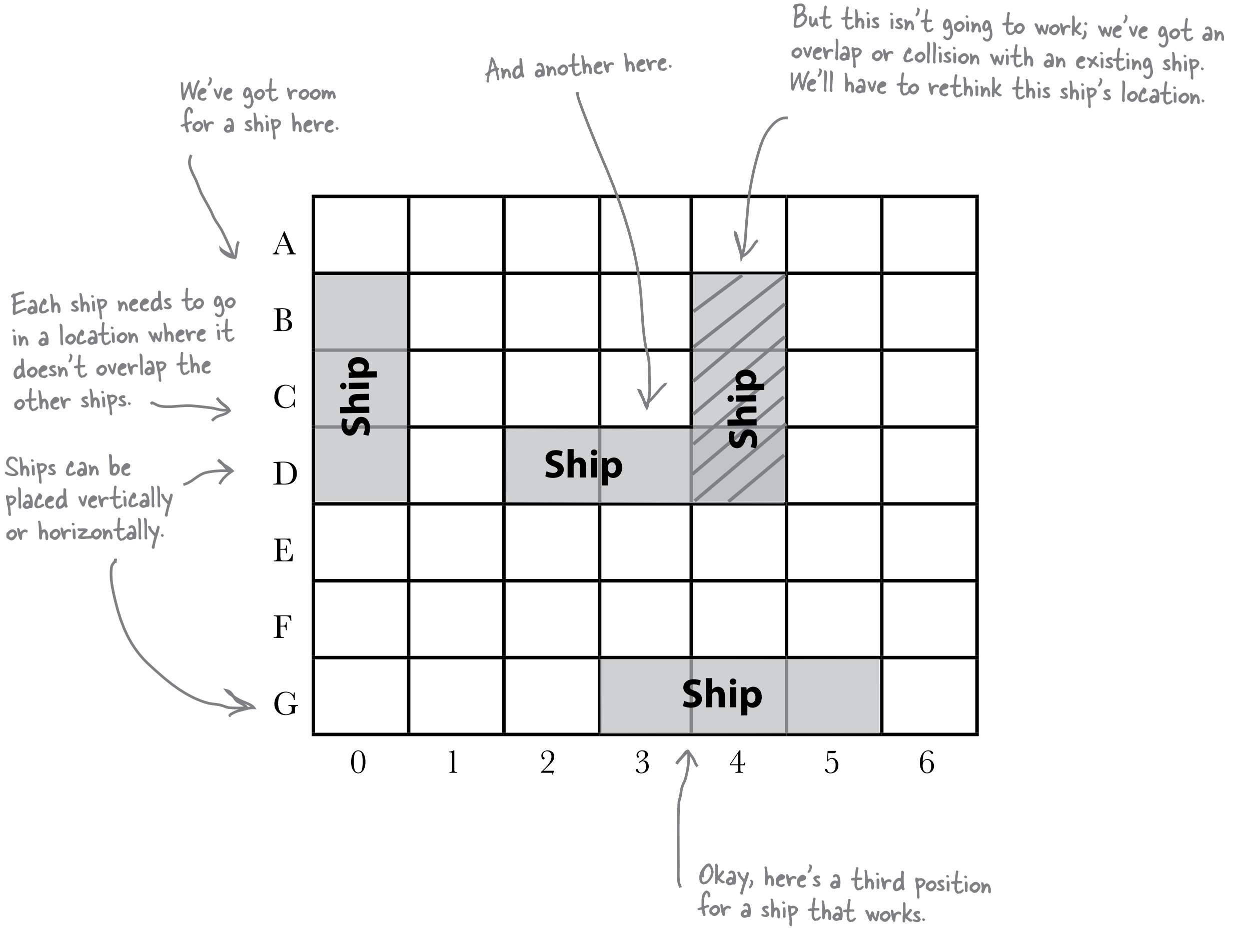
And we return false so the form doesn't do anything else (like try to submit itself).

Update your init function and add the handleKeyPress function anywhere in your code. Reload and let the gameplay begin!

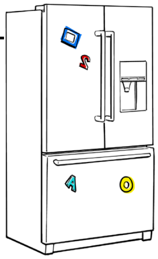
What's left? Oh yeah, darn it, those hardcoded ships!

At this point you've got a pretty amazing browser-based game created from a little HTML, some images, and roughly 100 lines of code. But the one aspect of this game that is a little unsatisfying is that the ships are always in the same locations. We still need to write the code to generate random locations for the ships every time we start a new game (otherwise, it'll be a pretty boring game).

Now, before we begin, we want to let you know that we're going to cover this code at a slightly faster clip—you're getting to the point where you can read and understand code better, and there aren't a lot of new things in this code. So, let's get started. Here's what we need to consider:



We've got room for a ship here.
And another here.
But this isn't going to work; we've got an overlap or collision with an existing ship. We'll have to rethink this ship's location.
Each ship needs to go in a location where it doesn't overlap the other ships.
Ships can be placed vertically or horizontally.
Okay, here's a third position for a ship that works.
A
B
C
D
E
F
G
0
1
2
3
4
5
6
Ship
Ship
Ship
Ship



Code Magnets

An algorithm to generate ships is all scrambled up on the fridge. Can you put the magnets back in the right places to produce a working algorithm? Check your answer at the end of the chapter before you go on.

An algorithm is just a fancy word for a sequence of steps that solve a problem.

```
Generate a random
location for the new ship.
```

```
Loop for the number of
ships we want to create.
```

```
Generate a random direction
(vertical or horizontal)
for the new ship.
```

```
Add the new ship's locations
to the ships array.
```

```
Test to see if the new ship's
locations collide with any
existing ship's locations.
```

Solution on page 395

How to place ships

There are two things you need to consider when placing ships on the game board. The first is that ships can be oriented either vertically or horizontally. The second is that ships can't overlap on the board. The bulk of the code we're about to write handles these two constraints. Now, as we said, we're not going to go through the code in gory detail, but you have everything you need to work through it, and if you spend enough time with the code you'll understand each part in detail. There's nothing in it that you haven't already encountered so far in the book (with one exception that we'll talk about). So let's dive in…

We're going to organize the code into three methods that are part of the model object:

- **generateShipLocations**: This is the master method. It creates a `ships` array in the model for you, with the number of ships in the model's `numShips` property.
- **generateShip**: This method creates a single ship, located somewhere on the board. The locations may or may not overlap with those of other ships.
- **collision**: This method takes a single ship and makes sure it doesn't overlap with a ship already on the board.

The generateShipLocations function

Let's get started with the `generateShipLocations` method. This method iterates, creating ships, until it has filled the model's `ships` array with enough ships. Each time it generates a new ship (which it does using the `generateShip` method), it uses the `collision` method to make sure there are no overlaps. If there is an overlap, it throws that ship away and keeps trying.

One thing to note in this code is that we're using a new iterator, the **do while** loop. The do while loop works almost exactly like **while**, except that you *first* execute the statements in the body, and *then* check the condition. You'll find certain logic conditions, while rare, work better with do while than with the while statement.

We're adding this method to the model object.

```
generateShipLocations() {
    let locations;
    for (let i = 0; i < this.numShips; i++) {
        do {
            locations = this.generateShip();
        } while (this.collision(locations));
        this.ships[i].locations = locations;
    }
},
```

For each ship we want to generate locations for...

...we generate a new set of locations...

We're using a do while loop here!

...and check to see if those locations overlap with any existing ships on the board. If they do, then we need to try again. We keep generating new locations until there's no collision.

Once we have locations that work, we assign the locations to the ship's locations property in the model.ships array.

Writing the generateShip method

The `generateShip` method creates an array with random locations for one ship without worrying about overlap with other ships on the board. We'll go through this method in a couple of steps. The first step is to randomly pick a direction for the ship: will it be horizontal or vertical? We're going to determine this with a random number. If the number is 1, then the ship is horizontal; if it's 0, then the ship is vertical. We'll use our friends the `Math.random` and `Math.floor` methods to do this, as we've done before:

Generating a random 0 or 1 is kind of like tossing a coin.

This method also is added to the model object.

```
generateShip() {
    let direction = Math.floor(Math.random() * 2);
    let row;
    let col;
    if (direction === 1) {
        // Generate a starting location for a horizontal ship
    } else {
        // Generate a starting location for a vertical ship
    }

    let newShipLocations = [];
    for (let i = 0; i < this.shipLength; i++) {
        if (direction === 1) {
            // add location to array for new horizontal ship
        } else {
            // add location to array for new vertical ship
        }
    }
    return newShipLocations;
},
```

We use Math.random to generate a number between 0 and 1, and multiply the result by 2 to get a number between 0 and 2 (not including 2). We then turn that into a 0 or a 1 using Math.floor.

We're saying that if direction is 1, that means we'll create a horizontal ship...

...and if direction is 0, that means we'll create a vertical ship.

First, we'll create a starting location, like row = 0 and column = 3, for the new ship. Depending on the direction, we need different rules to create the starting location (you'll see why in just a sec).

For the new ship locations, we'll start with an empty array and add the locations one by one.

We'll loop for the number of locations in a ship...

...and add a new location to the newShipLocations array each time through the loop. Again, we need slightly different code to generate a location depending on the direction of the ship.

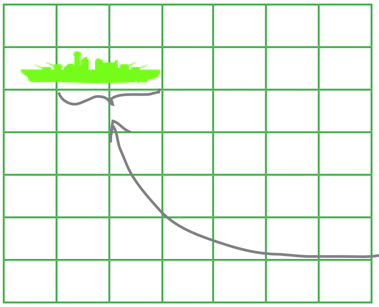
Once we've generated all the locations, we return the array.

We'll be filling in the rest of this code starting on the next page...

Generating the starting location for the new ship

Once we know how the ship is oriented, we can generate the locations for the ship. First, we'll generate the starting location (the first position for the ship); then, the rest of the locations will be in the next two columns (if the ship is horizontal) or the next two rows (if it's vertical).

So, we need to generate two random numbers—a row and a column—for the starting location of the ship. The numbers both have to be between 0 and 6, so the ship will fit on the game board. But remember, if the ship is going to be placed *horizontally*, then the starting *column* must be between 0 and 4, so that we have room for the rest of the ship:



A horizontal ship can be located in any row...

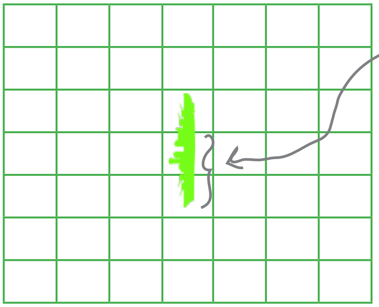
```
row = Math.floor(Math.random() * this.boardSize);
col = Math.floor(Math.random() * ((this.boardSize - 3) + 1));
```

...but the first column must leave room for the other two locations of the ship.

We're computing col in a generalized way so we can change the board size later if we want. We subtract 3 from the board size (7), and then add 1 back to that to get 5, so when we multiply by a random number between 0 and 1 we get a result that is between 0 and 4, not 0 and 3.

(Also, remember that boardSize is a property of the model.)

And likewise, if the ship is going to be placed *vertically*, then the starting *row* must be between 0 and 4, so that we have room for the rest of the ship:



A vertical ship must start at row 0-4 to leave room for the next two locations...

```
row = Math.floor(Math.random() * ((this.boardSize - 3) + 1));
col = Math.floor(Math.random() * this.boardSize);
```

...but can be located in any column.

Completing the generateShip method

Plugging that code in, now all we have to do is make sure we add the starting location along with the next two locations to the `newShipLocations` array:

```
generateShip() {
    let direction = Math.floor(Math.random() * 2);
    let row;
    let col;
    if (direction === 1) {
        row = Math.floor(Math.random() * this.boardSize);
        col = Math.floor(Math.random() * (this.boardSize - (this.shipLength + 1)));
    } else {
        row = Math.floor(Math.random() * (this.boardSize - (this.shipLength + 1)));
        col = Math.floor(Math.random() * this.boardSize);
    }
```

Here's the code to generate a starting location for the ship on the board.

We use this.shipLength (instead of 3, as on the previous page) to generalize the code, so we can use it for any ship length.

```
    let newShipLocations = [];
    for (let i = 0; i < this.shipLength; i++) {
        if (direction === 1) {
```

This is the code for a horizontal ship. Let's break it down...

Here, we use parentheses to make sure i is added to col before it's converted to a string.

```
            newShipLocations.push(`${row}${(col + i)}`);
```

We're pushing a new location onto the newShipLocations array.

That location is a string made up of the row (the starting row we just computed above)...

...and the column + i. The first time through the loop, i is 0, so it's just the starting column. The second time, it's the next column over, and the third, the next column over again. So, we'll get something like "01", "02", "03" in the array.

```
        } else {
            newShipLocations.push(`${(row + i)}${col}`);
```

Same thing here, only for a vertical ship.

So now, we're increasing the row instead of the column, adding i to the row each time through the loop.

For a vertical ship, we'll get something like "31", "41", "51" in the array.

```
        }
    }
    return newShipLocations;
},
```

Once we've filled the array with the ship's locations, we return it to the calling method, generateShipLocations.

Avoiding a collision!

The `collision` method takes a ship and checks to see if any of the locations overlap—or collide—with any of the existing ships already on the board.

Look back a few pages to see where we call the collision method in generateShipLocations.

We've implemented this using two nested for loops. The outer loop iterates over all the ships in the model (in the `model.ships` property). The inner loop iterates over all the new ship's locations in the `locations` array and checks to see if any of those locations is already taken up by an existing ship on the board:

locations is an array of locations for a new ship we'd like to place on the board.

```
collision(locations) {
    for (let i = 0; i < this.numShips; i++) {
        let ship = this.ships[i];
        for (let j = 0; j < locations.length; j++) {
            if (ship.locations.includes(locations[j])) {
                return true;
            }
        }
    }
    return false;
}
```

For each ship already on the board...

...check to see if any of the locations in the new ship's locations array are in an existing ship's locations array.

We're using includes to check if the location already exists in a ship. If includes returns true, we know it matched an existing location, so we return true (meaning, we found a collision).

Returning from inside a loop that's inside another loop stops the iteration of both loops immediately, exiting the function and returning true.

If we get here and haven't returned, then we never found a match for any of the locations we were checking, so we return false (there was no collision).

Brain Power

In this code, we have two loops: an outer loop to iterate over all the ships in the model, and an inner loop to iterate over each of the locations we're checking for a collision. For the outer loop, we used the loop variable i, and for the inner loop, we used the loop variable j. Why did we use two different loop variable names?

Two final changes

We've written all the code we need to generate random locations for the ships; now all we have to do is integrate it. Make these two final changes to your code, and then take your new Battleship game for a test drive!

```
let model = {
    boardSize: 7,
    numShips: 3,
    shipLength: 3,
    shipsSunk: 0,
    ~~ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },~~
             ~~{ locations: ["24", "34", "44"], hits: ["", "", ""] },~~
             ~~{ locations: ["10", "11", "12"], hits: ["", "", ""] } ],~~
    ships: [ { locations: [0, 0, 0], hits: ["", "", ""] },
             { locations: [0, 0, 0], hits: ["", "", ""] },
             { locations: [0, 0, 0], hits: ["", "", ""] } ],
    fire(guess) { ... },
    isSunk(ship) { ... },
    generateShipLocations() { ... },
    generateShip() { ... },
    collision(locations) { ... }
};

function init() {
    let fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
    let guessInput = document.getElementById("guessInput");
    guessInput.onkeypress = handleKeyPress;

    model.generateShipLocations();
}
```

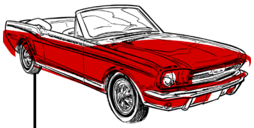
Remove the hardcoded ship locations...

...and replace them with arrays initialized with zeros instead.

And of course, add the call to generate the ship locations, which will fill in those empty arrays in the model.

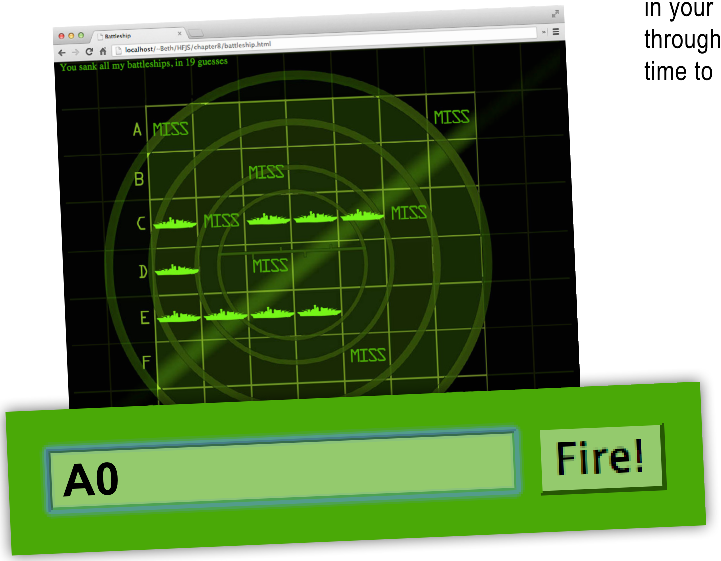
We're calling model.generateShipLocations from the init function so it happens right when you load the game, before you start playing. That way all the ships will have locations ready to go when you start playing.

Don't forget you can download the complete code for the Battleship game at http://wickedlysmart.com/hfjs.



Test Drive

This is the FINAL test drive of the real game, with random ship locations. Make sure you've got all the code added to "battleship.js", reload "battleship.html" in your browser, and play the game! Give it a good run through. Play it a few times, reloading the page each time to generate new ship locations for each new game.

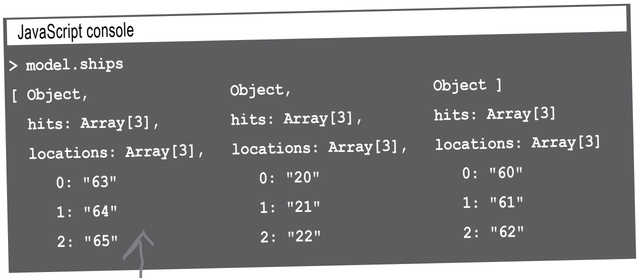

Battleship
localhost/~Beth/HFJS/chapter8/battleship.html

You sank all my battleships, in 19 guesses

A
B
C
D
E
F
MISS
MISS
MISS
MISS
MISS
MISS
MISS
MISS
A0
Fire!

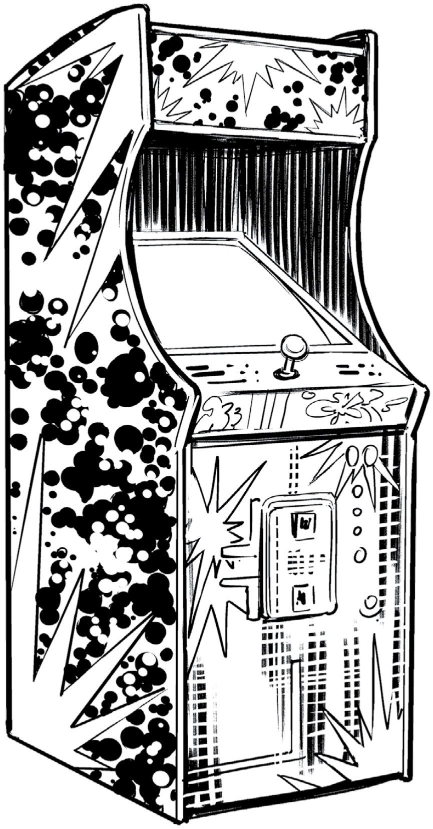
Oh, and how to cheat!

To cheat, open up the developer console and type `model.ships`. Press Return, and you should see the three ship objects containing the locations and hits arrays. Now you have the inside scoop on where the ships are sitting on the game board. But you didn't hear this from us!


JavaScript console

```
> model.ships
[ Object,                    Object,                    Object ]
  hits: Array[3],            hits: Array[3],            hits: Array[3]
  locations: Array[3],       locations: Array[3],       locations: Array[3]
     0: "63"                    0: "20"                    0: "60"
     1: "64"                    1: "21"                    1: "61"
     2: "65"                    2: "22"                    2: "62"
```

Beat the computer every time.



Congrats, it's startup time!

You've just built a great web application, all in 150 (or so) lines of code and some HTML and CSS. Like we said, the code is yours. Now all that's standing between you and your venture capital is a real business plan. But then again, who ever let that stand in their way!?

So now, after all the hard work, you can relax and play a few rounds of Battleship. Pretty darn engaging, right?

We went through a lot of code in this chapter. Be sure to have some good food and plenty of get rest to let it all sink in. But before you do that, you've got some bullet points to review and a crossword puzzle to do. Don't skip them; repetition is what really drives the learning home!

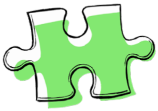
QA Notes

Found a bug, I think. When I get a hit on a ship, if I enter that board location again, it results in another hit at the same spot. Try it, you'll see!

So, is this a bug? Or the correct behavior? Assuming it's a bug, how would you fix it? Write your notes here:

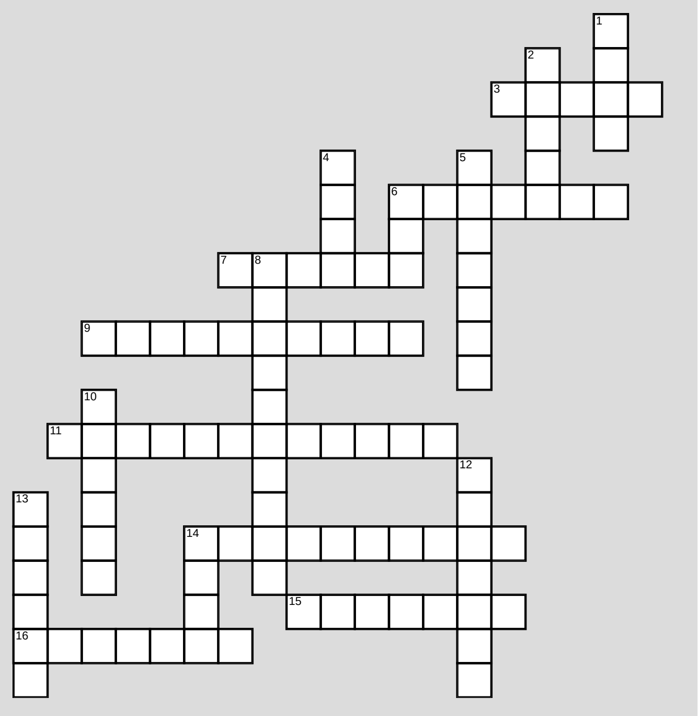
Bullet Points

- We use HTML to build the structure of the Battleship game, CSS to style it, and JavaScript to create the behavior.
- The id of each <td> element in the table is used to update the image of the element to indicate a HIT! or a MISS.
- The form uses an input with type “button”. We attach an **event handler** to the button so we can know in the code when a player has entered a guess.
- To get a value from a form input text element, use the element’s **value** property.
- CSS positioning can be used to position elements precisely in a web page.
- We organized the code using three objects: a **model**, a **view**, and a **controller**.
- Each object in the game has one **primary responsibility**.
- The responsibility of the model is to store the state of the game and implement logic that modifies that state.
- The responsibility of the view is to update the display when the state in the model changes.
- The responsibility of the controller is to glue the game together; it makes sure the player’s guess is sent to the model to update the state and checks to see when the game is complete.
- Designing the game with objects that each have **separate responsibilities** allows us to build and test each part of the game independently.
- To make it easier to create and test the model, we initially hardcoded the locations of the ships. After ensuring the model was working, we replaced these hardcoded locations with random locations generated by code.
- We used properties in the model, like numShips and shipLength, to avoid hardcoding values in the methods that we might want to change later.
- Arrays have an **includes** method that is similar to the string includes method. The array includes method takes a value and returns true if the array contains that value, and false otherwise.
- Arrays have an **indexOf** method that is similar to the string indexOf method. The array indexOf method takes a value and returns the index of that value if it exists in the array, or –1 if it does not.
- With **chaining**, you can string together object references (using the dot operator), thus combining statements and eliminating temporary variables.
- The **do while** loop is similar to the while loop, except that the condition is checked after the statements in the body of the loop have executed once.
- **Quality assurance** (QA) is an important part of developing your code. QA requires testing not just valid input, but invalid input as well.



JavaScript cross

Your brain is frying from the coding challenges in this chapter. Do the crossword to get that final sizzle.



ACROSS

3. The _______ notifies the view when its state changes.

6. To get the guess from the form input, we added an event _______ for the click event.

7. We represent each ship in the game with an _______.

9. Modern, interactive web apps use HTML, CSS, and _______.

11. We use the _______ method to set the class of an element.

14. The _______ is good at gluing things together.

15. The _______ loop executes the statements in its body at least once.

16. You can cheat and get the answers to Battleship using the _______.

DOWN

1. The three objects in our game design are the model, _______, and controller.

2. Who is responsible for state?

4. We call the _______ method to ask the model to update the state with the guess.

5. Arrays have an _______ method too.

6. To add a “hit” to the game board in the display, we add the _______ class to the corresponding <td> element.

8. To add a ship or MISS image to the board, we place the image in the _______ of a <td> element.

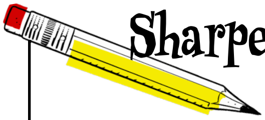
10. 13 is the keycode for the _______ key.

12. The responsibility of the collision function is to make sure that ships don’t _______.

13. Chaining is for _______ references, not just jailbirds.

14. The id of a <td> element corresponds to a _______ on the game board.

→ Solution on page 392



Sharpen your pencil Solution

From page 341

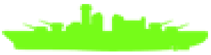
Before we write the code that's going to place hits and misses on the game board, get a little more practice to see how the CSS works. Manually play the game by adding the "hit" and "miss" classes into your markup, as dictated by the player's moves below. Imagine you've got ships at these locations:

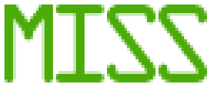
```
Ship 1: A6, B6, C6
Ship 2: C4, D4, E4
Ship 3: B0, B1, B2
```

and here are the player's guesses:

```
A0, D4, F5, B2, C5, C6
```

Make sure you've downloaded everything you need, including the two images you'll need for this exercise.





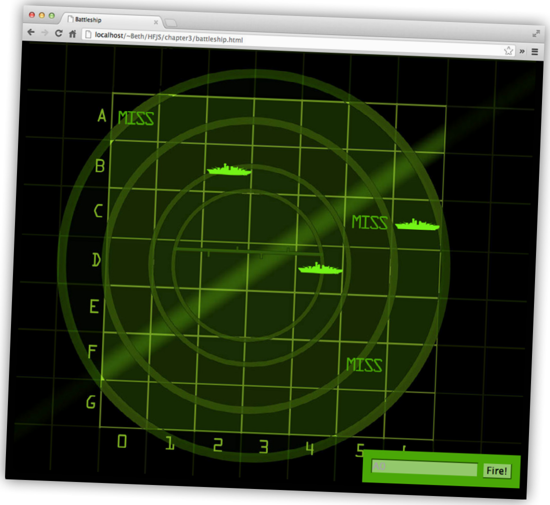
You need to add one of the two classes below to the correct cells in the grid (the correct <td> elements in the table) so that your grid shows MISS and a ship in the right places.

```
.hit {
    background: url("ship.png") no-repeat center center;
}
.miss {
    background: url("miss.png") no-repeat center center;
}
```

Here's our solution. The right spots for the .hit class are in <td>s with the ids "34", "12", and "26", and the ids for the .miss class are "00", "25", and "55". To add a class to an element, you use the class attribute, like this:

```
<td class="miss" id="55">
```

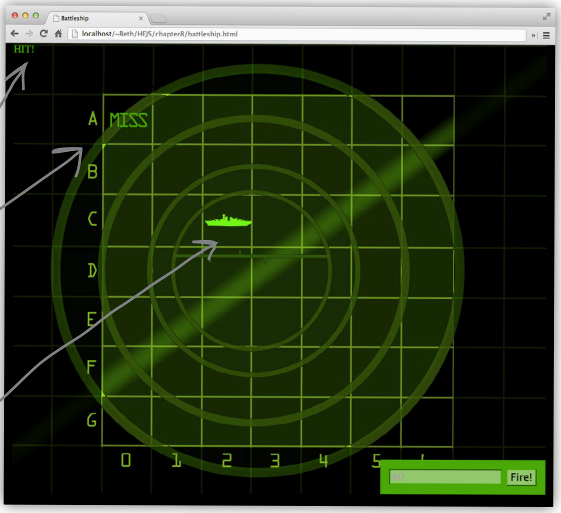
After adding the classes in the right spots, your game board should look like this.



Exercise Solution

From page 344

It's time for some object design. We're going to start with the view object. Now remember, the view object is responsible for updating the view. Take a look at the view below and see if you can determine the methods we want the view object to implement. Write down the declarations for these methods (just the declarations; we'll code the bodies of the methods in a bit), along with a comment or two about what each does. Here's our solution.



Here's a message. Messages will be things like "HIT!" or "You missed." or "You sank my battleship!"

Here, the display has a MISS placed on the grid.

And here, the display has a ship placed on the grid.

Notice we're defining an object and assigning it to the variable view.

Your methods go here!

```
let view = {

   // this method takes a string message and displays it
   // in the message display area
   displayMessage(msg) {
       // code to be supplied in a bit!
   },

   displayHit(location) {
       // code will go here
    },

    displayMiss(location) {
       // code will go here
    }
};
```

From page 351

Sharpen your pencil Solution

Given how we've described the new game board, how would you represent the ships in the model? (Just the locations; we'll worry about hits later.) Check off the best solution below.

- ☐ Use nine variables for the ship locations, similar to the way we handled the ships in Chapter 2.
- ☐ Use an array with an item for each cell in the board (49 items total). Record the ship number in each cell that holds part of a ship.
- ☐ Use an array to hold all nine locations. Items 0–2 will hold the first ship, 3–5 the second, and so on.
- ☐ Use three different arrays, one for each ship, with three locations contained in each.
- ☑ Use an object named ship with three location properties. Put all the ships in an array named ships.
- ☐ ______________________________
  ____
  ____

Or write in your own answer.

Any of these solutions could work! (In fact, we tried each one when we were figuring out the best way to do it.) This is the one we use in the chapter.

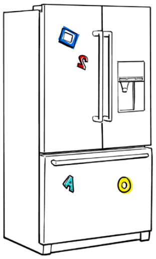
From page 375

JavaScript cross Solution

Your brain is frying from the coding challenges in this chapter. Do the crossword to get that final sizzle.

Here's our solution.

																	1 V	
															2 M		I	
														3 M	O	D	E	L
															D		W	
									4 F				5 I		E			
									I		6 H	A	N	D	L	E	R	
									R		I		D					
						7 O	8 B	J	E	C	T		E					
							A						X					
		9 J	A	V	A	S	C	R	I	P	T		O					
							K						F					
		10 R					G											
	11 S	E	T	A	T	T	R	I	B	U	T	E						
		T					O						12 O					
13 O		U					U						V					
B		R			14 C	O	N	T	R	O	L	L	E	R				
J		N			E		D						R					
E					L			15 D	O	W	H	I	L	E				
16 C	O	N	S	O	L	E							A					
T													P					



From page 353

Ship Magnets Solution

Use the following player moves, along with the data structure for the ships, to place the ship and MISS magnets onto the game board. Does the player sink all the ships?

Here are the moves:

A6, B3, C4, D1, B0, D4, F0, A1, C6, B1, B2, E4, B6

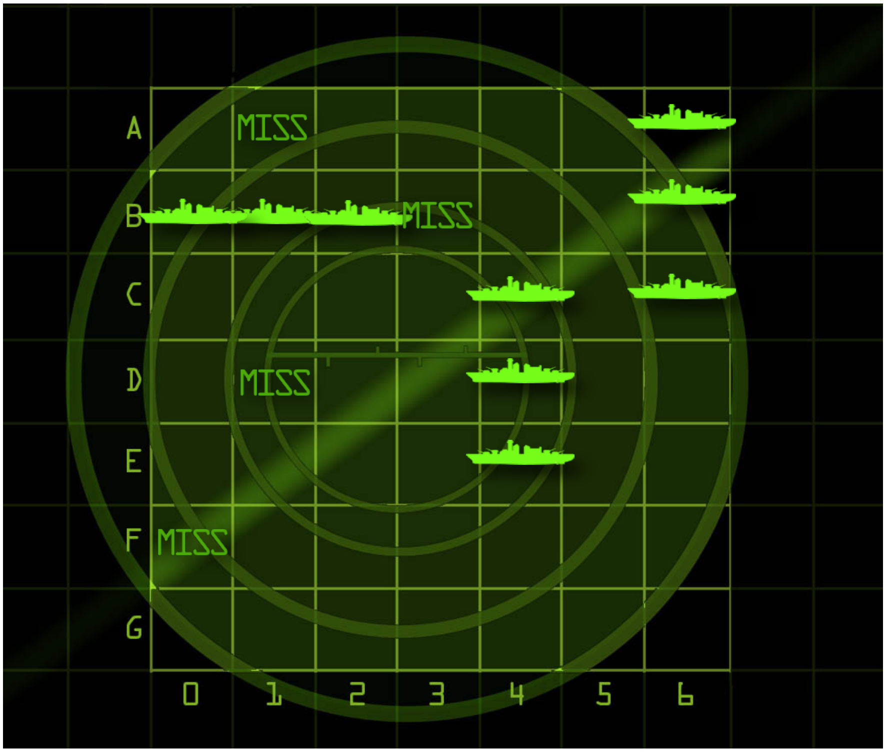
Execute these moves on the game board.

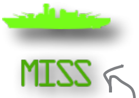
And here's our solution:

```
let ships = [{ locations: ["06", "16", "26"], hits: ["hit", "hit", "hit"] },
             { locations: ["24", "34", "44"], hits: ["hit", "hit", "hit"] },
             { locations: ["10", "11", "12"], hits: ["hit", "hit", "hit"] }];
```

All three ships are sunk!

And here's the board and your magnets.





Leftover magnets.

From page 354

Sharpen your pencil Solution

Let's practice using the ships data structure to simulate some ship activities. Using the ships definition below, work through the following questions and code and fill in the blanks. Here's our solution.

```
let ships = [{ locations: ["31", "41", "51"], hits: ["", "", ""] },
             { locations: ["14", "24", "34"], hits: ["", "hit", ""] },
             { locations: ["00", "01", "02"], hits: ["hit", "", ""] }];
```

Which ships are already hit?_Ships 2 and 3_ And at what locations? _C4, A0_

The player guesses "D4"; does that hit a ship?_yes_ If so, which one? _Ship 2_

The player guesses "B3"; does that hit a ship?_no_ If so, which one? __________

Finish this code to access the second ship's middle location and print its value with console.log:

```
let ship2 = ships[_1_];
let locations = ship2.locations;
console.log("Location is " + locations[_1_]);
```

Finish this code to see if the third ship has a hit in its first location:

```
let ship3 = ships[_2_];
let hits = ship3._hits_;
if (_hits[0]_ === "hit") {
   console.log("Ouch, hit on third ship at location one");
}
```

Finish this code to hit the first ship at the third location:

```
let _ship1_ = ships[0];
let hits = ship1._hits_;
hits[_2_] = _"hit"_;
```

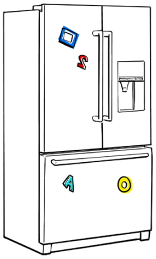
Sharpen your pencil Solution

From page 360

To see how isSunk works, work through these examples. Hint: includes returns true if the array contains the argument anywhere in the array; otherwise, it returns false (just like with strings). Here's our solution.

`ship.hits`	`ship.hits.includes("")`	`!ship.hits.includes("")`
`["", "", ""]`	true	false
`["", "hit", ""]`	true	false
`["hit", "hit", "hit"]`	false	true

The ship is sunk!



Code Magnets Solution

From page 378

An algorithm to generate ships is all scrambled up on the fridge. Can you put the magnets back in the right places to produce a working algorithm? Here's our solution.

```
Loop for the number of
ships we want to create.

    Generate a random direction
    (vertical or horizontal)
    for the new ship.

    Generate a random
    location for the new ship.

    Test to see if the new ship's
    locations collide with any
    existing ship's locations.

    Add the new ship's locations
    to the ships array.
```

Exercise Solution

From page 371

Here's our solution for moving parseGuess into the controller object.

Here, we've moved parseGuess into the controller and formatted it correctly to be a method of controller instead of a separate function.

```
let controller = {
    guesses: 0,

    parseGuess(guess) {
        const alphabet = ["A", "B", "C", "D", "E", "F", "G"];

        if (guess === null || guess.length !== 2) {
            alert("Oops, please enter a letter and a number on the board.");
        } else {
            let firstChar = guess.charAt(0);
            let row = alphabet.indexOf(firstChar);
            let column = guess.charAt(1);

            if (isNaN(row) || isNaN(column)) {
                alert("Oops, that isn't on the board.");
            } else if (row < 0 || row >= model.boardSize ||
                    column < 0 || column >= model.boardSize) {
                alert("Oops, that's off the board!");
            } else {
                return row + column;
            }
        }
        return null;
    },

    processGuess(guess) {
        let location = this.parseGuess(guess);
        if (location) {
            this.guesses++;
            let hit = model.fire(location);
            if (hit && model.shipsSunk === model.numShips) {
                view.displayMessage(`You sank all my battleships, in ${this.guesses}
guesses`);
            }
        }
    }
};
```

In processGuess, we just made one small change to refer to this.parseGuess instead of just parseGuess. That's it! Now our code is better organized.

9 handling events

Getting asynchronous

After this chapter, you're going to realize you aren't in Kansas anymore. Up until now, you've been writing code that typically executes from top to bottom—sure, your code might be a little more complex than that, and make use of a few functions, objects, and methods, but at some point the code just runs its course. Now, we're awfully sorry to break this to you this late in the book, but that's **not how you typically write JavaScript code**. Rather, most JavaScript is written to **react to events**. What kind of events? Well, how about a user clicking on your page, data arriving from the network, timers expiring in the browser, changes happening in the DOM...and that's just a few examples. In fact, all kinds of events are happening **all the time**, behind the scenes, in your browser. In this chapter we're going rethink our approach to JavaScript coding, and look at how and why we should write code that reacts to events.

Brain Power

You know what a browser does, right? It retrieves a page and all that page's contents, and then renders the page. But the browser's doing a lot more than just that. What else is it doing? Choose any of the tasks below you suspect the browser is doing behind the scenes. If you aren't sure, just make your best guess.

- ☐ Knows when the page is fully loaded and displayed
- ☐ Keeps track of all clicks made on the page, be they on a button, link, or elsewhere
- ☐ Knows when a user submits a form
- ☐ Knows when the user presses keys on the keyboard
- ☐ Knows when an element gets user interface focus
- ☐ Watches all mouse movement
- ☐ Watches the clock and manages timers and timed events
- ☐ Retrieves additional data for your page
- ☐ Tracks when the page has been resized or scrolled
- ☐ Knows when the cookies are finished baking

Sharpen your pencil

Pick two of the events from the Brain Power exercise above. If the browser could notify your code when these events occurred, what cool or interesting code might you write?

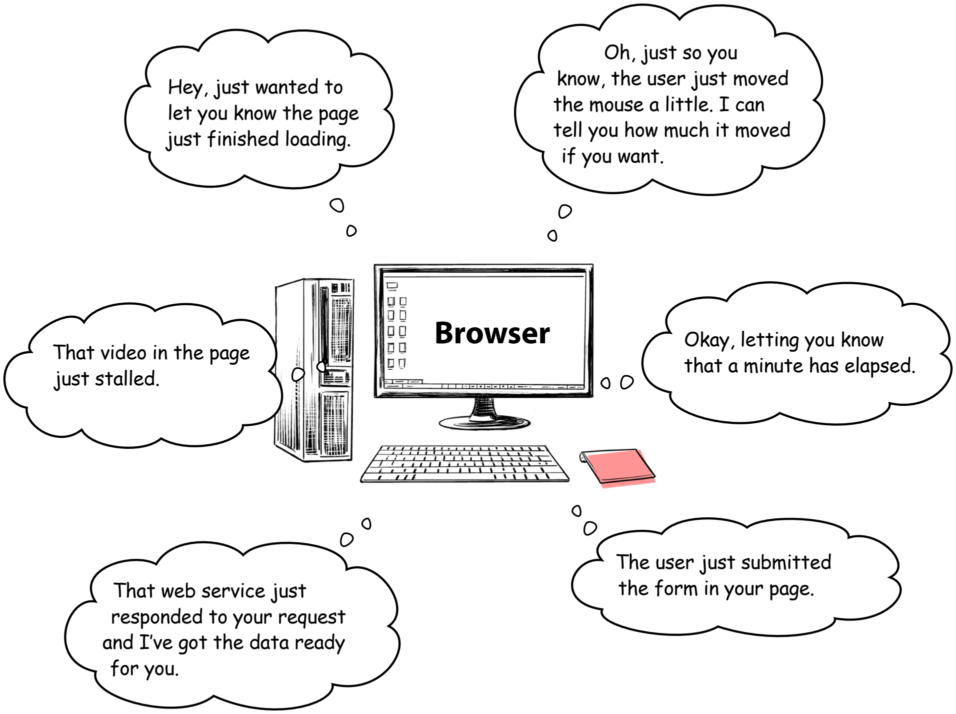
No, you can't use the cookie event as one of your examples!

Solution on page 437

What are events?

We're sure you know by now that after the browser retrieves and displays your page, it doesn't just sit there. Behind the scenes, a lot is going on: users are clicking buttons, the mouse location is being tracked, additional data is becoming available on the network, windows are getting resized, timers are going off, the window's location could be changing, and so on. All these things cause *events* to be triggered.

Whenever there's an event, there is an opportunity for your code to *handle it*; that is, to supply some code that will be invoked when the event occurs. Now, you're not required to handle any of these events, but you'll need to handle them if you want interesting things to happen when they occur—like, say, when the button click event happens you might want to add a new song to a playlist, or when new data arrives you might want to process it and display it on your page, or when a timer fires you might want to tell a user the hold on a front-row concert ticket is going to expire.


Hey, just wanted to let you know the page just finished loading.
Oh, just so you know, the user just moved the mouse a little. I can tell you how much it moved if you want.
That video in the page just stalled.
Browser
Okay, letting you know that a minute has elapsed.
That web service just responded to your request and I've got the data ready for you.
The user just submitted the form in your page.

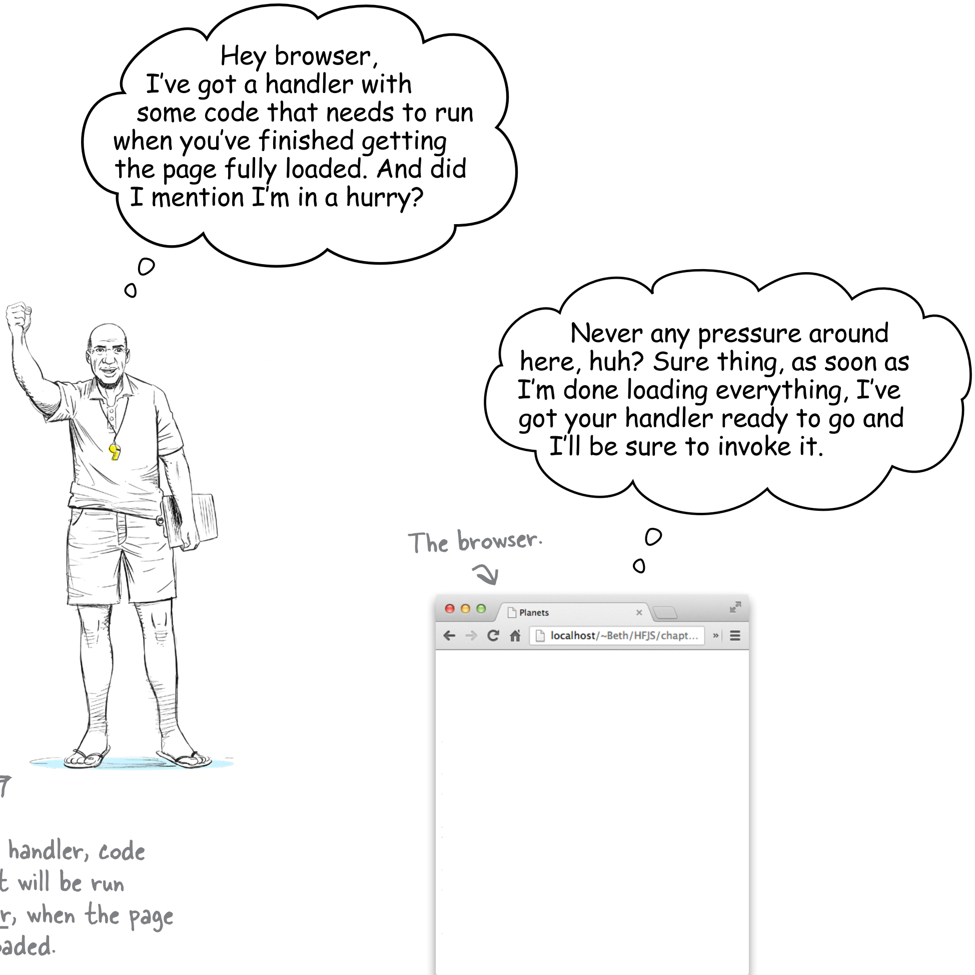
Whenever there's an event, there is an opportunity for your code to handle it.

What's an event handler?

We write *handlers* to handle events. Handlers are typically small pieces of code that know what to do when an event occurs. In terms of code, a handler is just a function. When an event occurs, its handler function is called.

You might also hear developers use the name callback or listener instead of handler.

To have your handler called when an event occurs, you first need to *register it*. As you'll see, there are a few different ways to do that, depending on what kind of event it is. We'll get into all that, but for now let's get started with a simple example, one you've seen before: the event that's generated when a page is fully loaded.


Hey browser, I've got a handler with some code that needs to run when you've finished getting the page fully loaded. And did I mention I'm in a hurry?
Never any pressure around here, huh? Sure thing, as soon as I'm done loading everything, I've got your handler ready to go and I'll be sure to invoke it.
Planets
localhost/~Beth/HFJS/chapt...

The browser.

The handler, code that will be run later, when the page is loaded.

Creating an event handler

There's no better way to understand events than by writing a handler and wiring it up to handle a real, live event. Now, remember, you've already seen a couple of examples of handling events—including the page load event—but we've never fully explained how event handling works. The page load event is triggered when the browser has fully loaded and displayed all the content in your page (and built out the DOM representing the page).

Let's step through what it takes to write the handler and to make sure it gets invoked when the page load event is triggered:

1. First, we need to write a function that can handle the page load event when it occurs. In this case, the function is going to announce to the world "I'm alive!" when it knows the page is fully loaded.

A handler is just an ordinary function.

```
function pageLoadedHandler() {
    alert("I'm alive!");
}
```

Here's our function; we'll name it pageLoadedHandler, but you can call it anything you like.

Remember, we often refer to this as a handler or a callback.

This event handler doesn't do much. It just creates an alert.

2. Now that we have a handler written and ready to go, we need to wire things up so the browser knows there's a function it should invoke when the load event occurs. To do that, we use the onload property of the window object, like this:

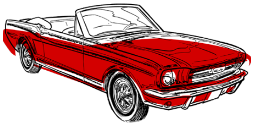
```
window.onload = pageLoadedHandler;
```

In the case of the load event, we assign a reference to the handler to the window's onload property.

Now, when the page load event is generated, the pageLoadedHandler function is going to be called.

We're going to see that different kinds of events are assigned handlers in different ways.

3. That's it! Now, with this code written, we can sit back and know that the browser will invoke the function assigned to the window.onload property when the page is loaded.



Test Drive

Go ahead and create a new file, “event.html”, and add the code to test your load event handler. Load the page into the browser and make sure you see the alert.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> I'm alive! </title>
  <script>
    window.onload = pageLoadedHandler;
    function pageLoadedHandler() {
        alert("I'm alive!");
    }
  </script>
</head>
<body>
</body>
</html>
```

First, the browser loads your page and starts parsing the HTML and building up the DOM.

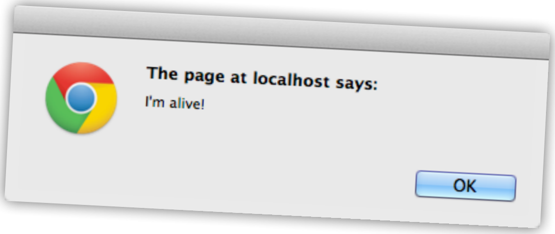
When it gets to your script, the browser starts executing the code.

For now, the script just defines a function and assigns that function to the window.onload property. Remember, this function will be invoked when the page is fully loaded.

Then, the browser continues parsing the HTML.

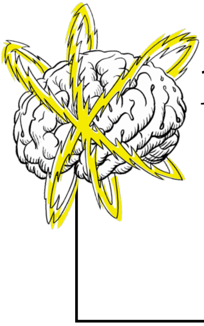
When the browser is done parsing the HTML, and the DOM is ready, the browser calls the page load handler.

Which in this case creates the “I'm alive!” alert.



The page at localhost says:

I'm alive!



Brain Power

If we didn’t have functions, could we have event handlers?

If you're ever going to be a real JavaScript developer, you're going to have to learn to deal with events.

As we've already mentioned, up until now you've taken a rather, let's say, linear approach to writing code: you took an algorithm, like computing the best bubble solution, or generating the 99 bottles of rootbeer lyrics, and wrote the code stepwise, top to bottom.

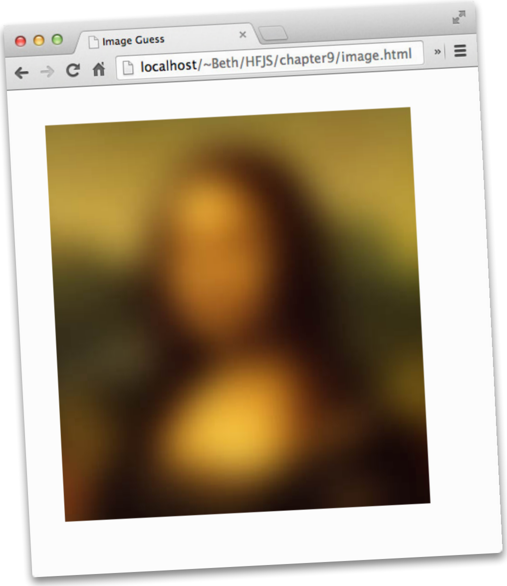
But remember the Battleship game? The code for that game didn't quite fit the linear model. Sure, you wrote some code that set up the game, initialized the model, and all that—but then the main part of the game operated in a *different way*. Each time you wanted to fire at another ship, you entered your guess into a form input element and pressed the "Fire" button. That button then caused a whole sequence of actions that resulted in the next move of the game being executed. In that case, your code was *reacting* to the user input.

Organizing code around reacting to events is a different way of thinking about how you write your code. To write code this way, you need to consider the events that can happen, and how your code should react. Computer science types like to say that this kind of code is *asynchronous*, because we're writing code to be invoked *later*, *if* and *when* an event occurs. This kind of coding also changes your perspective from one of encoding an algorithm step-by-step into code into one of gluing together an application that is composed of many handlers handling many different kinds of events.

Getting your head around events...by creating a game

The best way to understand events is with experience, so let's get some more by writing a simple game. The game works like this: you load a page and are presented with an image. Not just any image, but a really blurred image. Your job is to guess what the image is. And, to check your answer, you click on the image to unblur it.

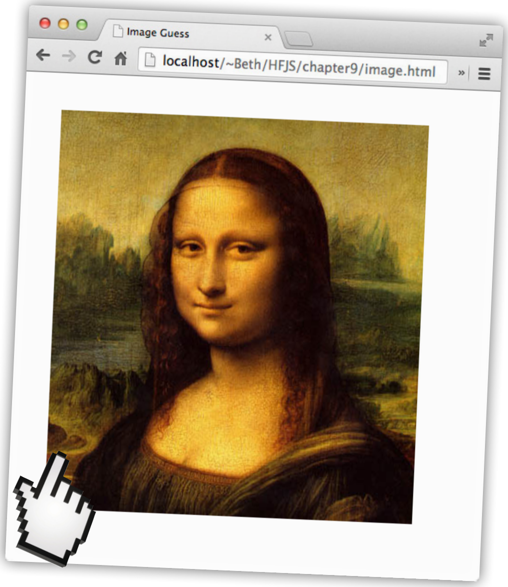
Like this:



Here's the blurred version of the image. Hmmm, what could it be?

Hum the Jeopardy theme song to yourself as you try to figure it out...

When you think you have it, click to reveal the unblurred image.



Let's start with the markup. We'll use two JPG images. One is blurred, and the other isn't. We've named them "zeroblur.jpg" and "zero.jpg" respectively. Here's the markup:

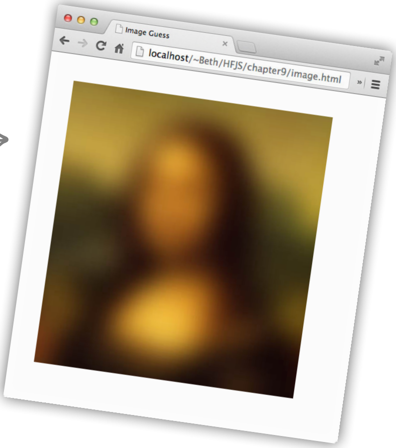
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style> body { margin: 20px; } </style>
  <script>  </script>
</head>
<body>
    <img id="zero" src="zeroblur.jpg">
</body>
</html>
```

Just some basic HTML, with a <script> element all ready for our code. Rather than use a separate file for the JavaScript, we'll keep it simple and add the script here. As you'll see, there is very little code needed to implement this.

And here's the blurred image, placed in the page. We'll give it an id of "zero". You'll see how we use the id in a sec...

Implementing the game

Go ahead and load this markup in your browser, and you'll see the blurred image. To implement the game, we need to react to a click on the image in order to display the unblurred version of the image.



Lucky for us, every time an HTML element in the page is clicked (or touched on a mobile device), an event is generated. Your job is to create a handler for that event, and in it write the code to display the unblurred version of the image. Here's how you're going to do that:

1. **Access the image object in the DOM and assign a handler to its onclick property.**
2. **In your handler, write the code to change the image src attribute from the blurred image to the unblurred one.**

Let's walk through these steps and write the code.

Step 1: Access the image in the DOM

Getting access to the image is old hat for you; we just need to use our old friend, the `getElementById` method, to get a reference to it:

```
let image = document.getElementById("zero");
```

Here we're grabbing a reference to the image element and assigning it to the image variable.

Oh, but we also need this code to run only *after* the DOM for the page has been created, so let's use the window's `onload` property to ensure that. We'll place our code into a function, `init`, that we'll assign to the `onload` property:

Remember, we can't get the image from the DOM until the page has finished loading.

```
window.onload = init;
function init() {
    let image = document.getElementById("zero");
}
```

We create a function init, and assign it to the window's onload property to make sure this code doesn't run until the page is fully loaded.

In the code of init, we'll grab a reference to the image with id="zero".

Remember, in JavaScript the order in which you define your functions doesn't matter. So, we can define init after we assign it to the onload property.

Step 2: Add the handler and update the image

To add a handler to deal with clicks on the image, we simply assign a function to the image's `onclick` property. Let's call that function `showAnswer` (we'll define it next):

```
window.onload = init;
function init() {
    let image = document.getElementById("zero");
    image.onclick = showAnswer;
}
```

Using the image object from the DOM, we're assigning a handler to its onclick property.

Now we need to write the `showAnswer` function, which unblurs the image by resetting the image element's `src` property to the unblurred image:

First, we have to get the image from the DOM again.

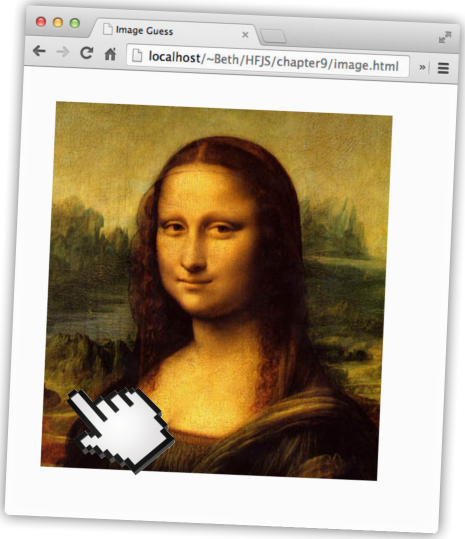
```
function showAnswer() {
    let image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Remember, the blurred version is named "zeroblur.jpg" and the unblurred one is named "zero.jpg".

Once we have the image, we can change it by setting its src property to the unblurred image.

Test drive

Let's take this simple game for a test drive. Make sure you've got all the HTML, CSS, and JavaScript typed into a file named "image.html", and that you've got the images you downloaded from http://wickedlysmart.com/hfjs in the same folder. Once all that's done, load up the file in your browser and give it a try!



Click anywhere on the image to have the showAnswer handler called. When that happens, the src of the image is changed to reveal the answer.

Wait, why do we need to use getElementById again in showAnswer? I'm not sure of the flow of execution here.

Ah, yes. It can get tricky to follow the flow of execution in code with a lot of event handlers. Remember, the `init` function is called when the page is loaded. But the `showAnswer` function isn't called until later, when you click the image. So, these two event handlers get called at two different times.

In addition, remember your scope rules. In the `init` function we're putting the object returned by `getElementById` into a *local* variable `image`, which means when that function completes, the variable falls out of scope and is destroyed. So later, when the `showAnswer` function is called, we have to get the image object again from the DOM. Sure, we could have put this in a global variable, but overuse of globals can lead to confusing and buggy code, which we'd like to avoid.

there are no Dumb Questions

Q: Is setting the src property of the image the same as setting the src attribute using setAttribute?

A: In this case, yes, it is. When you get an HTML element from the DOM using getElementById, you're getting an element object that has several methods and properties. All element objects come with a property, id, that is set to the id of the HTML element (if you've given it one in your HTML). The image element object also comes with a src property that is set to the image file specified in the src attribute of the <img> element.

Not all attributes come with corresponding object properties, however, so you will need to use setAttribute and getAttribute for those. And in the case of src and id, you can either use the properties or get/set them using getAttribute and setAttribute, and it does the same thing.

Q: So do we have a handler called within a handler?

A: Not really. The load handler is the code that is called when the page is fully loaded. When the load handler is called, we assign a handler to the image's onclick property, but it won't be called until you actually click on the image. When you do that (potentially a long time after the page has loaded), the showAnswer click handler is called. So, the two handlers get called at different times.

BE the Browser

Below, you'll find your game code. Your job is to play like you're the browser and figure out what you need to do after each event. After you've done the exercise, look at the solution at the end of the chapter to see if you got everything. We've done the first bit for you.

```
window.onload = init;
function init() {
    let image = document.getElementById("zero");
    image.onclick = showAnswer;
}

function showAnswer() {
    let image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Here's the code you're executing...

When page is being loaded...	First define the functions init and showAnswer
When page load event occurs...	
When image click event occurs...	

Your answers go here.

→ Solution on page 438

Brain Power

What if you had an entire page of images that could each be individually unblurred by clicking? How would you design your code to handle this? Make some notes. What might be the naive way of implementing this? Is there a way to implement this with minimal changes to the code you've already written?

Cubicle Conversation

Mara
Sam
Alex

Mara: So far, the image guessing game works great. But we really should expand the game to include more images on the page.

Sam: Sure, Mara, that's exactly what I was thinking.

Alex: Hey, I've already got a bunch of images ready to go, we just need the code. I've followed the naming convention of "zero.jpg", "zeroblur.jpg", "one.jpg", "oneblur.jpg", and so on...

Sam: Are we going to need to write a new click event handler for each image? That's going to be a lot of repetitive code. After all, every event handler's going to do exactly the same thing: replace the blurred image with its unblurred version, right?

Alex: That's true. But I'm not sure I know how to use the same event handler for multiple images. Is that even possible?

Mara: What we can do is assign the same handler, which really means the same function, to the `onclick` property of every image in the game.

Alex: So the same function gets called for every image that is clicked on?

Mara: Right. We'll use `showAnswer` as the handler for every image's click event.

Sam: Hmm, but how will we know which image to unblur?

Alex: What do you mean? Won't the click handler know?

Sam: How will it know? Right now, our `showAnswer` function assumes we clicked on the image with the id "zero". But if we're calling `showAnswer` for every image's click event, then our code needs to work for any of the images.

Alex: Oh…right…so how do we know which image was clicked?

Mara: Actually, I've been reading up on events, and I think there is a way for the click handler to know the element the user clicked on. But let's deal with that part later. First, let's add some more images to the game and see how to set the same event handler for all of them...then we'll figure out how to determine which image the user clicked.

Alex, Sam: Sounds good!

Let's add some more images

We've got a whole set of new images, so let's start by adding them to the page. We'll add five more images, for a total of six. We'll also modify the CSS to add a little whitespace between the images:

Get the images

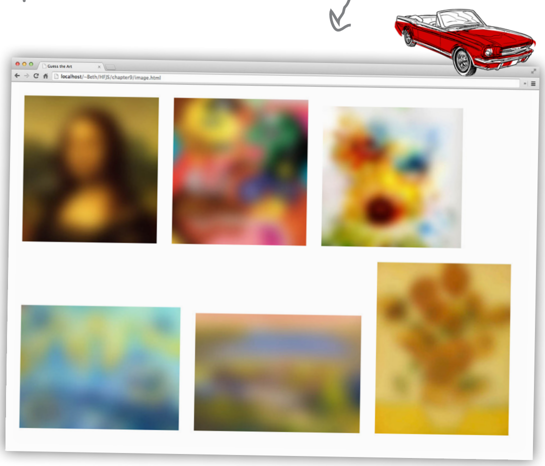
You'll find all the images in the chapter9 folder you downloaded from https://wickedlysmart.com/hfjs.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style>
    body { margin: 20px; }
    img { margin: 20px; }
  </style>
  <script>
     window.onload = init;
     function init() {
        let image = document.getElementById("zero");
        image.onclick = showAnswer;
    }
    function showAnswer() {
        let image = document.getElementById("zero");
        image.src = "zero.jpg";
    }
  </script>
</head>
<body>
    <img id="zero" src="zeroblur.jpg">
    <img id="one" src="oneblur.jpg">
    <img id="two" src="twoblur.jpg">
    <img id="three" src="threeblur.jpg">
    <img id="four" src="fourblur.jpg">
    <img id="five" src="fiveblur.jpg">
</body>
</html>
```

We're just adding a margin of 20px between the images with this CSS property.

And here are the five new images we're adding. Notice we're using the same id and src naming scheme (and image naming scheme) for each one. You'll see how this is going to work in a bit...

If you give this a quick test drive, your page should look like this:



Now we need to assign the same event handler to each image's onclick property

Now we have more images in the page, but we have more work to do. Right now you can click on the first image (of the *Mona Lisa*) and see the unblurred image, but what about the other images?

We *could* write a new, separate handler function for each image, but from the discussion so far, you know that would be tedious and wasteful. Here, have a look:

```
window.onload = init;
function init() {
    let image0 = document.getElementById("zero");
    image0.onclick = showImageZero;
    let image1 = document.getElementById("one");
    image1.onclick = showImageOne;
    ...
}
function showImageZero() {
    let image = document.getElementById("zero");
    image.src = "zero.jpg";
}
function showImageOne() {
    let image = document.getElementById("one");
    image.src = "one.jpg";
}
...
```

We could get each image element from the page and assign a separate click handler to each one. We'd have to do this six times... we're only showing two here.

The other four would be set here.

And we'd need six different click handlers, one for each image.

And we'd need four more handler functions here.

Brain Power

What are the disadvantages of writing a separate handler for each image? Check all that apply:

☐ Lots of redundant code in each handler.

☐ Generates a lot of code.

☐ Hard to generalize for an arbitrary number of images.

☐ If we need to change the code in one handler, we're probably going to have to change them all.

☐ Hard to keep track of all the images and handlers.

☐ Harder for others to work on the code.

How to reuse the same handler for all the images

Clearly, writing a handler for each image isn't a good way to solve this problem. So what we're going to do instead is use our existing handler, `showAnswer`, to handle all of the click events for all the images. Of course, we'll need to modify `showAnswer` a little bit to make this work. To use `showAnswer` for all the images, we need to do two things:

1. **Assign the showAnswer click handler function to every image on the page.**

2. **Rework showAnswer to handle unblurring any image, not just zero.jpg.**

And we'd like to do both these things in a generalized way that works even if we add more images to the page. In other words, if we write the code right, we should be able to add images to the page (or delete images from the page) without any code changes. Let's get started.

Assigning the click handler to all images on the page

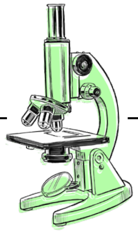
Here's our first hurdle: in the current code we use the `getElementById` method to grab a reference to image "zero", and assign the `showAnswer` function to its `onclick` property. Rather than hardcoding a call to `getElementById` for each image, we're going to show you an easier way: we'll grab all the images at once, iterate through them, and set up the click handler for each one. To do that we'll use a DOM method you haven't seen yet: `document.getElementsByTagName`. This method takes a tag name, like `img` or `p` or `div`, and returns a list of elements that match it. Let's put it to work:

```
function init() {
    ~~var image = document.getElementById("zero");~~
    ~~image.onclick = showAnswer;~~

    let images = document.getElementsByTagName("img");
    for (let i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};
```

We'll get rid of the old code to get image "zero" and set its handler.

Now we're getting elements from the page using a tag name, img. This finds every image in the page and returns them all. We store the resulting images in the images variable.

Then we iterate over the images and assign the showAnswer click handler to each image in turn. Now the onclick property of each image is set to the showAnswer handler.



document.getElementsByTagName Up Close

The `document.getElementsByTagName` method works a lot like `document.getElementById`, except that instead of getting an element by its id, we're getting elements by tag name—in this case, the tag name "img". Of course, your HTML can include many `<img>` elements, so this method may return many elements, or one element, or even zero elements, depending on how many images are in the page. In our image game example, we have six `<img>` elements, so we'll get back a list of six image objects.

What we get back is a list of element objects that match the specified tag name.

```
let images = document.getElementsByTagName("img");
```

What's returned is an array-*like* list of objects. It's not exactly an array but has qualities similar to an array.

Notice the "s" here. That means we might get many elements back.

Put the tag name in quotes here (and don't include the < and >!).

there are no Dumb Questions

Q: You said getElementsByTagName returns a list. Do you mean an array?

A: It returns an object that you can treat like an array, but it's actually an object called an HTMLCollection. An HTMLCollection is a collection of elements in the DOM tree with a given tag name. You can iterate over this collection by getting its length using the length property, and then access each item in the HTMLCollection using an index with the bracket notation, just like an array. But that's pretty much where the similarities of an HTMLCollection and an array end, so beyond this, you'll need to be careful in how you deal with the HTMLCollection object. You typically won't need to know more about HTMLCollection until you want to start adding and removing elements to and from the DOM.

Q: So I can assign a click handler to any element?

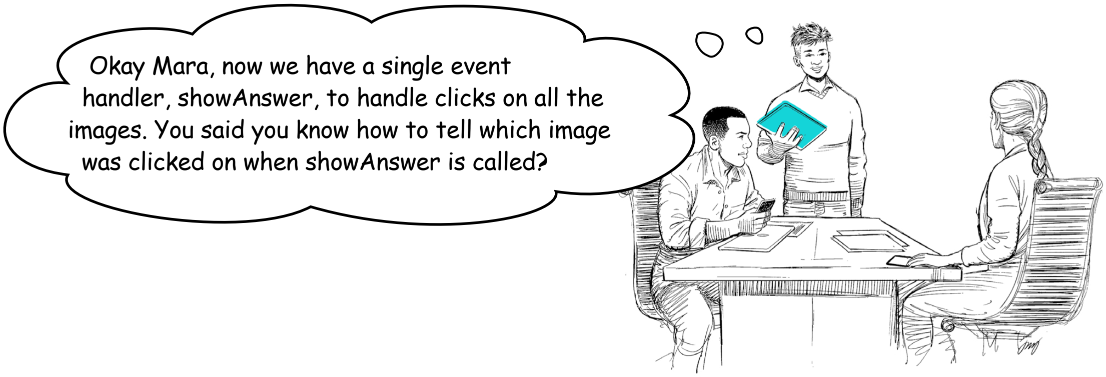
A: Pretty much. Take any element on the page, get access to it, and assign a function to its onclick property. Done. As you've seen, that handler might be specific to that one element, or you might reuse a handler for events on many elements. Of course, elements that don't have a visual presence in your page, like the <script> and <head> elements, won't support events like the click event.

Q: Do handler functions ever get passed any arguments?

A: Ah, good question, and very timely. They do, and we're just about to look at the event object that gets passed to some handlers.

Q: Do elements support other types of events? Or is the click the only one?

A: There are quite a few others. In fact, you've already seen another one in the code of the Battleship game: the keypress event. There, an event handler function was called whenever the user pressed Return from the form input. We'll take a look at a few other event types in this chapter.


Okay Mara, now we have a single event handler, showAnswer, to handle clicks on all the images. You said you know how to tell which image was clicked on when showAnswer is called?

Mara: Yes, I do. Whenever the click event handler is called, it's passed an *event object*. You can use that object to find out details about the event.

Alex: Like which image was clicked on?

Mara: Well, more generally, the element on which the event occurred, which is known as the *target*.

Alex: What's the target?

Mara: It's the element that generated the event. Like, if you click on a specific image, the target will be that image.

Alex: So if I click on the image with the id "zero", then the target will be set to that image?

Mara: More precisely, the element object that represents that image. Think of the element object as exactly the same thing you get if you call `document.getElementById` with a value of "zero". It's the object that represents the image in the DOM.

Alex: Okay, so how do we get this target? It sounds like that's what we need to know which image was clicked on.

Mara: The target is just a property of the event object.

Alex: Great. That sounds perfect for `showAnswer`. We'll be done in a snap…Wait, so `showAnswer` is passed the event object?

Mara: That's right.

Alex: So how did our code for the `showAnswer` function work up until now? It's being passed this event object, but we don't have a parameter defined for the event object in the function!

Mara: Remember, JavaScript lets you ignore arguments if you want.

Alex: Oh, right.

Mara: Now Alex, don't forget, you'll need to figure out how to change the `src` of the image to the correct name for the unblurred version. Right now we're assuming the name of the unblurred image is "zero.jpg", but that won't work anymore.

Alex: Maybe we can use the `id` attribute of the image to figure out the unblurred image name. The ids of all the images match the names of the unblurred version of each image. Let's try it!

How the event object works

When the click handler is called, it's passed an *event object*—and in fact, for most of the events associated with the document object model (DOM) you'll be passed an event object. The event object contains general information about the event, such as what element generated the event and what time the event happened. In addition, you'll get information specific to the event, so if there was a mouse click, for instance, you'll get the coordinates of the click.

There are other kinds of events too (that is, other than DOM events); we'll see an example later in the chapter.

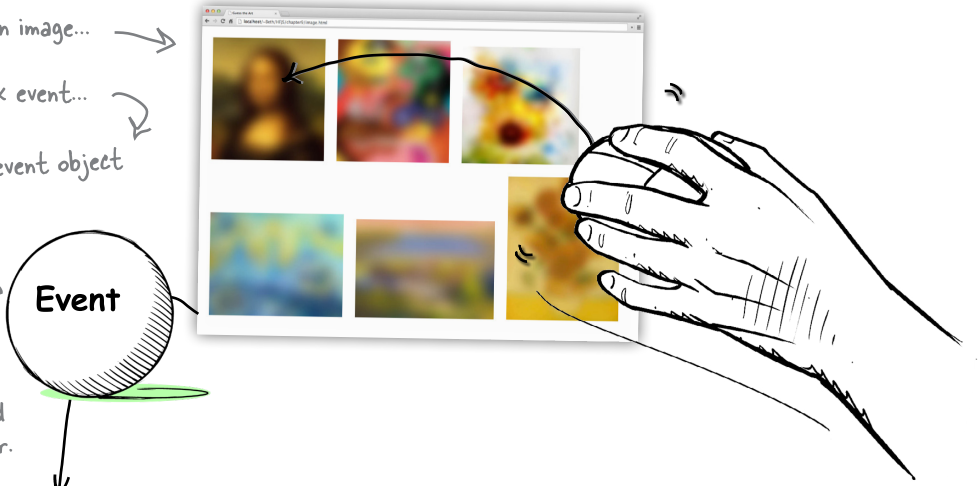
Let's step through how event objects work:

Take our game as an example.

You click on an image...

...which causes a click event...

...which causes an event object to be created...



...that is then passed to the event handler.

```
function showAnswer(eventObj) {
    ...
}
```

In the handler, you can use the event object to determine things about the event, like what type of event occurred, which element generated the event, and so on.

So, what is *in* an event object? Like we said, both general and specific information about the event. The specific information depends on the type of the event, and we'll come back to that a bit later. The general information includes the `target` property that holds a reference to the object that generated the event. So, if you click on a page element, like an image, that's the target, and you can access it like this:

```
function showAnswer(eventObj) {
    let image = eventObj.target;
}
```

The target tells us what element generated the event.

Who Does What?

You've already seen that the event object (for DOM events) has properties that give you more information about the event that just happened. Below, you'll find other properties that the event object can have. Match each event object property to what it does.

target

type

timeStamp

keyCode

x

y

touches

Want to know how far from the top of the browser window the user clicked? Use me.

I hold the object on which the event occurred. I can be different kinds of objects, but most often I'm an element object.

Using a touch device? Then use me to find out how many fingers are touching the screen.

I'm a string, like "click" or "load", that tells you what just happened.

Want to know when your event happened? I'm the property for you.

Want to know how far from the left side of the browser window the user clicked? Use me.

I'll tell you what key the user just pressed.

→ Solution on page 439

Putting the event object to work

Now that we've learned a little more about events—or more specifically, how the event object is passed to the click handler—let's figure out how to use the information in the event object to unblur any image on the page. We'll start by revisiting the HTML markup:

```
<!doctype html>
 ...
<body>
    <img id="zero" src="zeroblur.jpg">
    <img id="one" src="oneblur.jpg">
    <img id="two" src="twoblur.jpg">
    <img id="three" src="threeblur.jpg">
    <img id="four" src="fourblur.jpg">
    <img id="five" src="fiveblur.jpg">
</body>
</html>
```

Here's the HTML again.

Each of the images has an id, and the id corresponds to the unblurred image name. So, the image with id "zero" has an unblurred image of "zero.jpg", the image with id "one" has an unblurred image of "one.jpg", and so on...

Notice that the value of each image's id corresponds to the name of the unblurred image (minus the ".jpg" extension). If we can access this id, then we can simply take that name and add on ".jpg" to create the name of the corresponding unblurred image. Once we have that, we can change the image `src` property to the unblurred version of the image. Let's see how:

Remember, we're getting passed an event object each time an image is clicked on.

```
function showAnswer(eventObj) {
   let image = eventObj.target;

   let name = image.id;
   image.src = `${name}.jpg`;
}
```

The event object's target property is a reference to the image element that was clicked.

We can then use the id property of that object to get the name of the unblurred image.

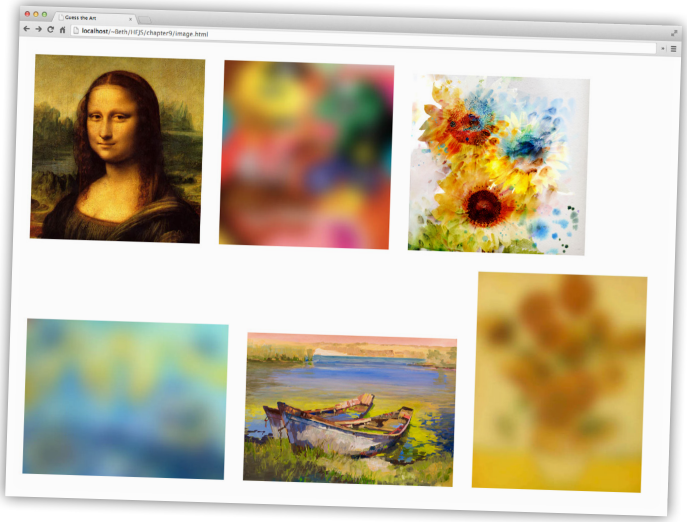
And finally, we'll set the src of the image to that name.

As you know, once you change the src property of the image, the browser will immediately retrieve that new image and display it in the page in place of the blurred version.

Test drive the event object and target

Make sure you've updated all the code in your "image.html" file, and take it for a test drive. Guess the image, click, and see the unblurred version revealed. Think about how this app is designed not as a top-to-bottom running program, but purely as a set of actions that result from an event being generated when you click on an image. Also think about how you handled the events on all the images with one piece of code that's smart enough to know which image was clicked on. Play around. What happens if you click on an image twice? Anything at all?

Now we can click on any of the images and see the unblurred version. How well did you do?



there are no Dumb Questions

Q: Does the onload event handler get passed an event object too?

A: It does, and it includes information like the target, which is the window object, the time it happened, and the type of the event, which is "load". It's safe to say you don't typically see the event object used much in load handlers, because there really isn't anything that is useful in it for this kind of event. You're going to find that sometimes the event object will be useful to you, and sometimes it won't, depending on the type of event. If you're unsure what the event object contains for a specific kind of event, just grab a JavaScript reference.

Brain Power

What if you want to have an image become blurred again a few seconds after you've revealed the answer? How might that work?

Events Exposed

This week's interview: Talking to the Browser about events

Head First: Hey Browser, it's always good to have your time. We know how busy you are.

Browser: My pleasure, and you're right, managing all these events keeps me on my toes.

Head First: Just how do you manage them, anyway? Give us a behind-the-scenes look at the magic.

Browser: As you know, events are almost continually happening. The user moves the mouse around, or makes a gesture on a mobile device; things arrive over the network; timers go off…it's like Grand Central. That's a lot to manage.

Head First: I would have assumed you don't need to do much unless there happens to be a handler defined somewhere for an event...

Browser: Even if there's no handler, there's still work to do. Someone has to grab the event, interpret it, and see if there is a handler waiting for it. If there is, I have to make sure that handler gets executed.

Head First: So how do you keep track of all these events? What if lots of events are happening at the same time? There's only one of you, after all.

Browser: Well, yes, lots of events can happen over a very short amount of time, sometimes too fast for me to handle all in real time. So what I do is throw them all on a queue as they come in. Then I go through the queue and execute handlers where necessary.

Head First: Boy, that sounds like my days as a short-order cook!

Browser: Sure, if you had orders coming in every millisecond or so!

Head First: You have go through the queue one by one?

Browser: I sure do, and that's an important thing to know about JavaScript: there's one queue and one "thread of control," meaning there is only one of me going through the events one at a time.

Head First: What does that really mean for our readers learning JavaScript?

Browser: Well, say you write a handler and it requires a lot of computation—that is, something that takes a long time to compute. As long as your handler is chugging along computing, I'm sitting around waiting until it's done. Only then can I continue with the queue.

Head First: Oh wow. Does that happen a lot—that is, you end up waiting on slow code?

Browser: It happens, but it also doesn't take long for a web developer to figure out the page or app isn't responsive because the handlers are slow. So, it's not a common problem as long as the web developers know how event queues work.

Head First: And now all our readers do! Now, back to events. Are there lots of different kinds of events?

Browser: There are. We've got network-based events, timer events, DOM events related to the page, and a few others. Some kinds of events, like DOM events, generate event objects that contain a lot more detail about the event—like, a mouse click event will have information about where the user clicked, and a keypress event will have information about which key was pressed, and so on.

Head First: So, you spend a lot of time dealing with events. Is that really the best use of your time? After all, you've got to deal with retrieving, parsing, and rendering pages, and all that.

Browser: Oh, it's very important. These days you've got to write code that makes your pages interactive and engaging, and for that you need events.

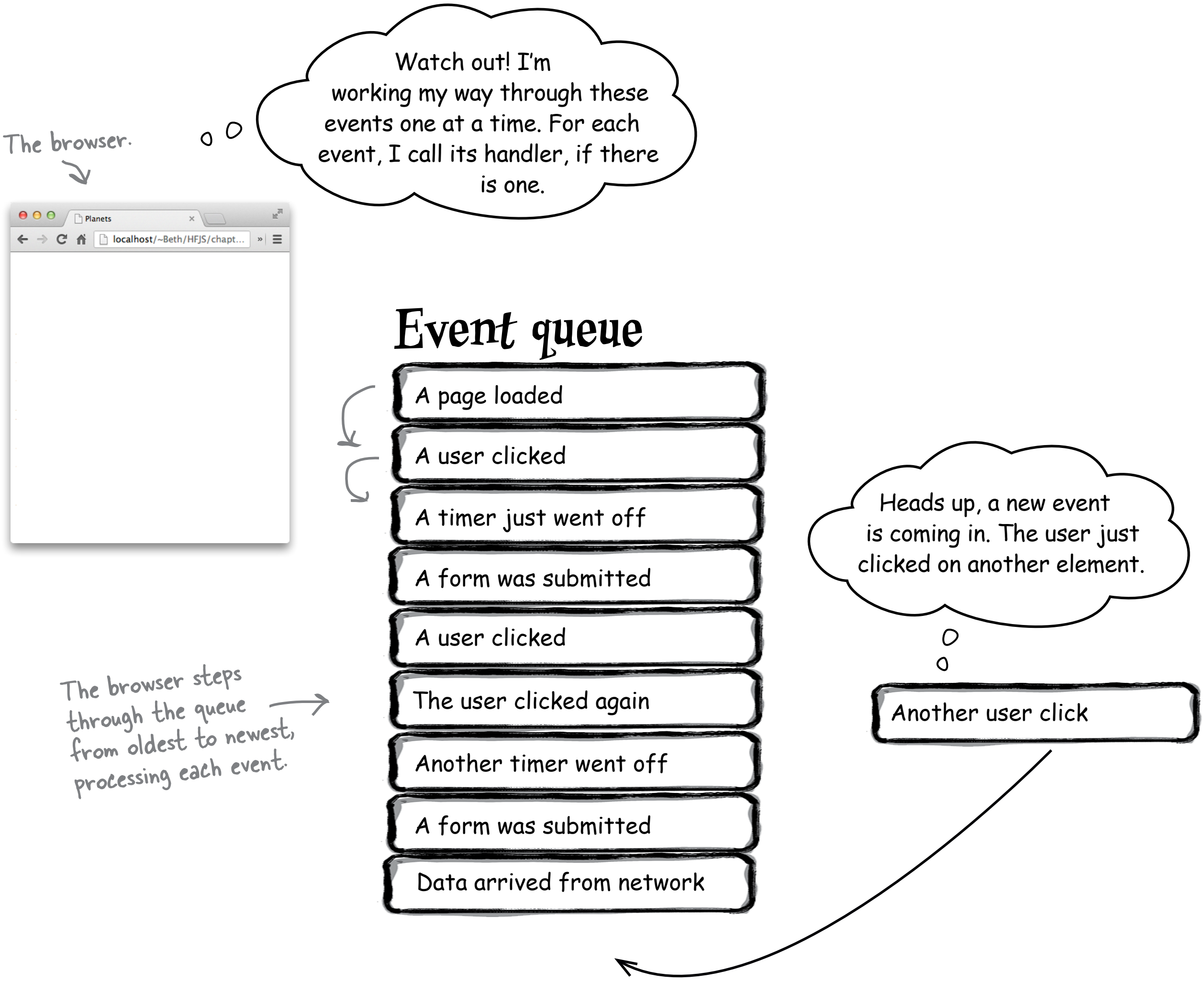
Head First: Oh for sure, the days of simple pages are gone.

Browser: Exactly. Oh shoot, this queue is about to overflow. Gotta run!

Head First: Okay…until next time!

Events and queues

You already know that the browser maintains a queue of events. And that behind the scenes the browser is constantly taking events off that queue and processing them by calling the appropriate event handler for them, if there is one.



The browser.

Watch out! I'm working my way through these events one at a time. For each event, I call its handler, if there is one.
Planets
localhost/~Beth/HFJS/chapt...
Event queue
A page loaded
A user clicked
A timer just went off
A form was submitted
A user clicked
The user clicked again
Another timer went off
A form was submitted
Data arrived from network
Heads up, a new event is coming in. The user just clicked on another element.
Another user click

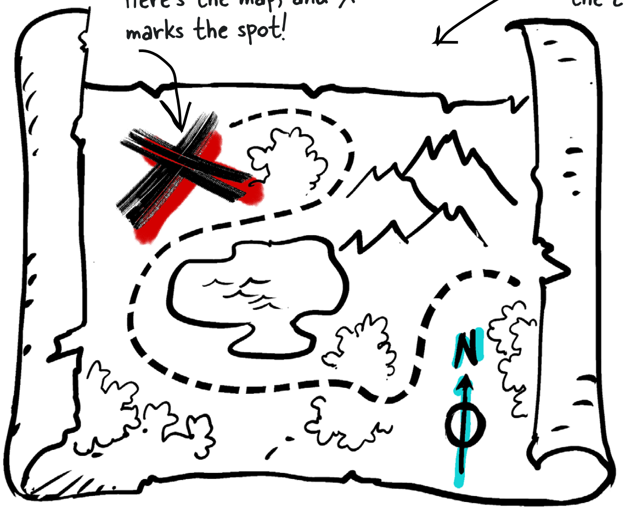
The browser steps through the queue from oldest to newest, processing each event.

It's important to know that the browser processes these events one at a time, so, where possible, you need to keep your handlers short and efficient. If you don't, the event queue could stack up with waiting events, and the browser will get backed up dealing with them all. The downside to you? Your interface could really start to become slow and unresponsive.

If things get really bad you'll get the slow script dialog box, which means the browser is giving up!

Exercise

Ahoy, matey! You've got a treasure map in your possession, and we need your help in determining the coordinates of the treasure. To do that, you're going to write a bit of code that displays the coordinates on the map as you pass the mouse over the map. We've got some of the code on the next page, but you'll have to help finish it.



Here's the map, and X marks the spot!

After your code is written, just move the mouse over the X to see the coordinates of the treasure.

Your code will display the coordinates below the map.

COORDINATES: 10, 20

P.S. We highly encourage you to do this exercise, because we don't think the pirates are going to be too happy if they don't get their coordinates...Oh, and you'll need this to complete your code.

The mousemove event

The mousemove event notifies your handler when a mouse moves over a particular element. You set up your handler using the element's **onmousemove** property. Once you've done that, you'll be passed an event object that provides these properties:

x, y: The x (and y) position in pixels of your mouse from the left side (and top) of the browser window

screenX, screenY: The x (and y) position in pixels of your mouse from the left side (and top) of the user's screen

pageX, pageY: The x (and y) position in pixels of your mouse from the left side (and top) of the browser's page

Exercise

Blimey! The code is below. So far, it includes the map in the page and creates a paragraph element to display the coordinates. You need to make all the event code work. Good luck. We don't want to see you go to Davy Jones's locker anytime soon...

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Pirates Booty</title>
  <script>
       window.onload = init;
       function init() {
           let map = document.getElementById("map");
           __________________________________________
       }

       function showCoords(eventObj) {
           let coords = document.getElementById("coords");
           ________________________
           ________________________
          coords.innerHTML = `Map coordinates: ${x}, ${y}`;
       }
  </script>
</head>
<body>
       <img id="map" src="map.jpg">
       <p id="coords">Move mouse to find coordinates...</p>
</body>
</html>
```

Set up your handler here.

Grab the coordinates here.

We use the coordinates here.

When you're done, get this code in a real page, load it, and write your coordinates here.

Solution on page 440

____

Even more events

So far, we've seen three types of events: the load event, which occurs when the browser has loaded the page; the click event, which occurs when a user clicks on an element in the page; and the mousemove event, which occurs when a user moves the mouse over an element. You're likely to run into many other kinds of events too, like events for data arriving over the network, events about the geolocation of your browser, and time-based events (just to name a few).

For all the events you've seen, to wire up a handler, you've always assigned the handler to some property, like `onload`, `onmouseover`, or `onclick`. But not all events work like this—for example, with time-based events, rather than assigning a handler to a property, you call a function, `setTimeout`, and pass it your handler.

As an example, say you want your code to wait five seconds before doing something. Here's how you'd do that using `setTimeout` and a handler:

First, we write an event handler. This is the handler that will be called when the time event has occurred.

```
function timerHandler() {
    alert("Hey what are you doing just sitting there staring at a blank screen?");
}
```

All we're doing in this event handler is showing an alert.

And here, we call setTimeout, which takes two arguments: the event handler and a time duration (in milliseconds).

```
setTimeout(timerHandler, 5000);
```

Using setTimeout is a bit like setting a stopwatch.

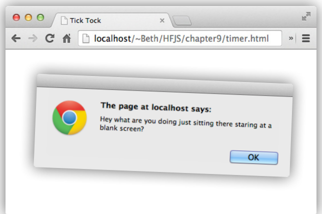
Here, we're asking the timer to wait 5,000 milliseconds (5 seconds).

And then we call the handler, timerHandler.

Test drive your timer

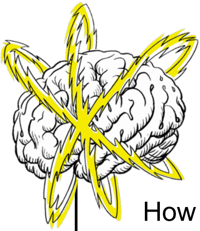
Don't just sit there! It's time to test this code! Throw the code into a basic HTML page, and load the page. At first you won't see anything, but after five seconds you'll see the alert.

Be patient, wait five seconds, and you'll see what we see. Now, if you've been sitting there a couple minutes you might want to give your machine a little kick...just kidding, you'd actually better check your code.



The page at localhost says:

Hey what are you doing just sitting there staring at a blank screen?



Brain Power

How would you make the alert appear every 5 seconds, over and over?

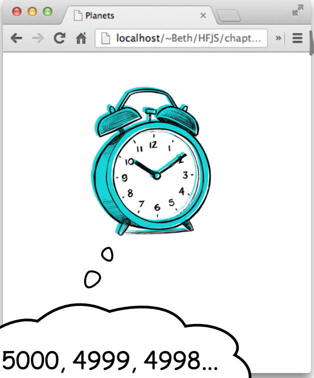
How setTimeout works

Let's step through what just happened:

Okay, it's go time. I've got a timer that goes off in 5,000 milliseconds, and I've got a handler to call when that happens.

Your browser manages timers.

1. When the page loads we do two things: we define a handler named timerHandler, and we call setTimeout to create a time event that will be generated in 5,000 milliseconds. When the time event happens, the handler will be executed.

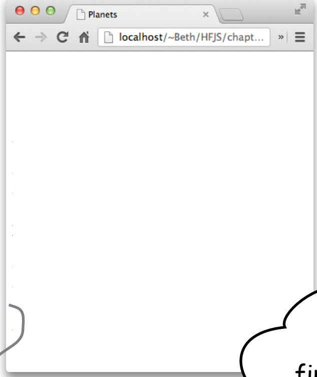

Planets
localhost/~Beth/HFJS/chapt...
5000, 4999, 4998...
..., 6, 5, 4, 3, 2, 1, 0.

The browser tracks all timers (yes, you can have more than one at a time) along with the corresponding handlers it needs to call.

2. The browser continues its normal job as the timer counts down in milliseconds.

That's 5,000 milliseconds; the timer is done, let's call that handler.

3. When the browser's countdown gets to zero, the browser calls the handler.

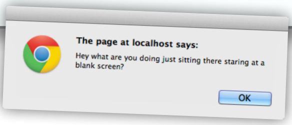

Planets
localhost/~Beth/HFJS/chapt...
The handler has been called. I'm finished with that timer.

The time event is triggered when the countdown is complete. The browser executes the event handler by calling the function you passed in.

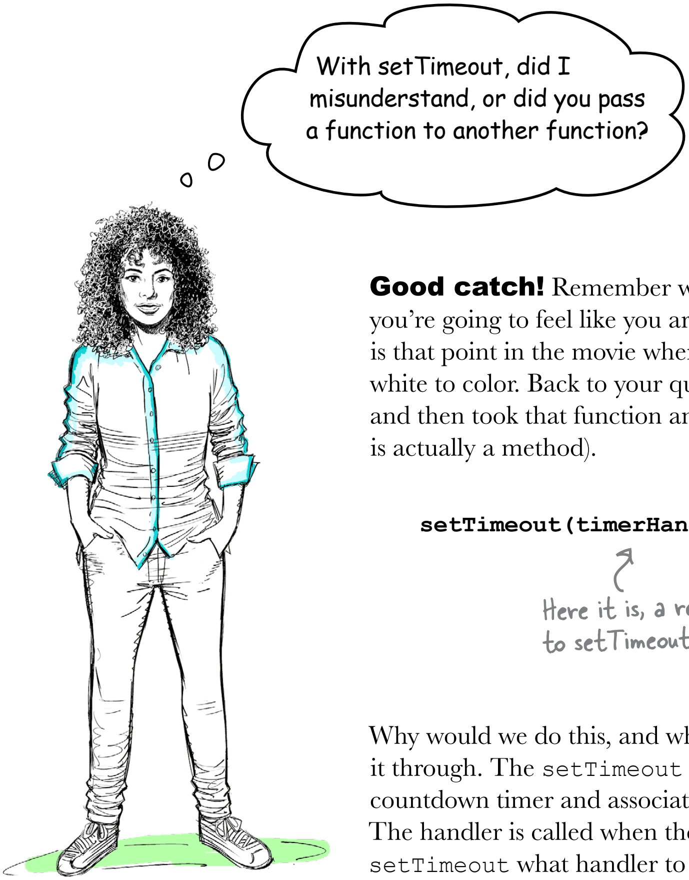
4. The handler is called, resulting in an alert being created and displayed in the browser.

```
function timerHandler() {
    alert("Hey what are you doing just sitting there staring at a blank screen?");
}
```

When the browser executes our event handler, we see the alert!


The page at localhost says:

Hey what are you doing just sitting there staring at a blank screen?


With setTimeout, did I misunderstand, or did you pass a function to another function?

Good catch! Remember we said up front that in this chapter you're going to feel like you aren't in Kansas anymore? Well, this is that point in the movie where everything goes from black and white to color. Back to your question: yes, we defined a function and then took that function and passed it to `setTimeout` (which is actually a method).

```
setTimeout(timerHandler, 5000);
```

Here it is, a reference to a function passed to setTimeout (another function).

Why would we do this, and what does it mean? Let's think it through. The `setTimeout` function essentially creates a countdown timer and associates a handler with that timer. The handler is called when the timer hits zero. Now, to tell `setTimeout` what handler to call, we need to pass it a *reference to the handler function*. `setTimeout` stores the reference away to use later, when the timer has expired.

If you're saying "That makes sense," then great. On the other hand, you might be saying "Excuse me? Pass a function to a function? Say what?" In that case, you probably have experience with a language where *you don't just go around passing functions to other functions like this*…Well, in JavaScript, you do, and in fact, being able to pass functions around is incredibly powerful, especially when we're writing code that reacts to events.

More likely, at this point you're saying "I think I sort of get it, but I'm not sure." If so, no worries. For now, just think of this as giving `setTimeout` a reference to the handler it's going to need to invoke when the timer expires. We're going to be talking a lot more about functions and what you can do with them (like passing them to other functions) in the next chapter. So just go with it for now.

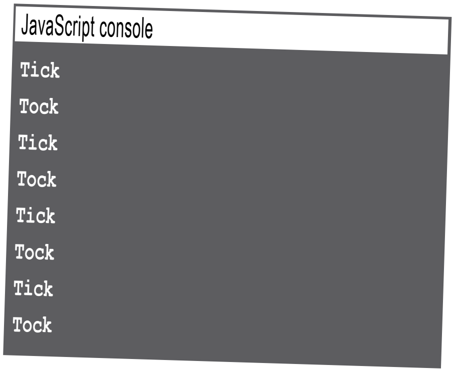
Exercise

Take a look at the code below and see if you can figure out what setInterval does. It's similar to setTimeout, but with a slight twist. Check your answer at the end of the chapter.

Here's the code.

```
let tick = true;
function ticker() {
    if (tick) {
        console.log("Tick");
        tick = false;
    } else {
        console.log("Tock");
        tick = true;
    }
}
setInterval(ticker, 1000);
```

Your analysis goes here.



```
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
```

Here's the output.

Solution on page 441

there are no Dumb Questions

Q: Is there a way to stop setInterval?

A: There is. When you call setInterval, it returns a timer object. You can pass that timer object to another function, clearInterval, to stop the timer.

Q: You said setTimeout was a method, but it looks like a function. Where's the object it's a method of?

A: Good catch. Technically, we could write window.setTimeout, but because the window object is considered the global object, we can omit the object name and just use setTimeout. You'll see this a lot in practice.

Q: Can I omit window on the window.onload property too?

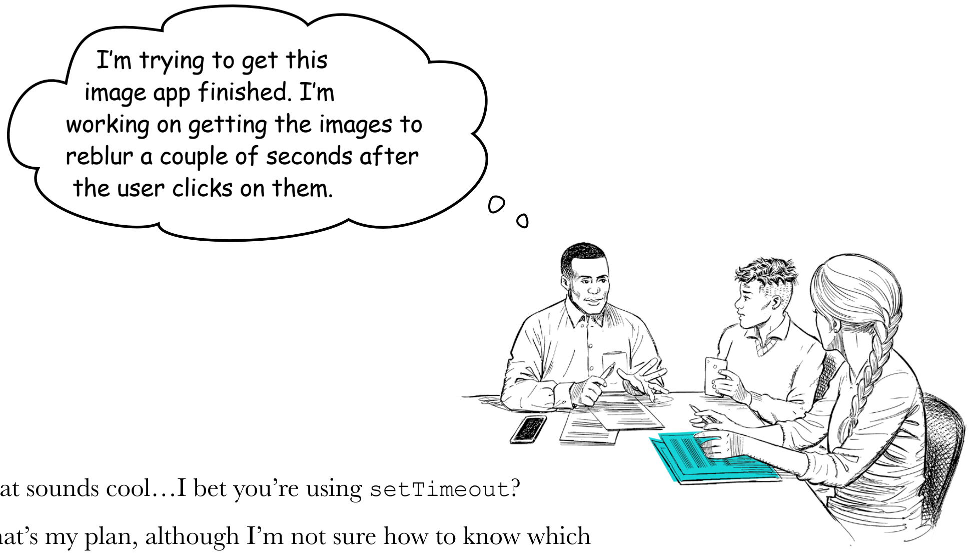
A: You can, but most people don't because they are worried onload is a common enough property name (other elements can have the onload property too) that not specifying which onload property they mean might be confusing.

Q: With onload, I'm assigning one handler to an event. But with setTimeout, I seem to be able to assign as many handlers as I want to as many timers as I want?

A: Exactly. When you call setTimeout, you are creating a timer and associating a handler with it. You can create as many timers as you like. The browser keeps track of associating each timer with its handler.

Q: Are there other examples of passing functions to functions?

A: Lots of them. In fact, you'll find that passing around functions is fairly common in JavaScript. Not only do lots of built-in functions, like setTimeout and setInterval, make use of function passing, but you'll also discover there's a lot code you'll write yourself that accepts functions as arguments. But that's only part of the story, and in the next chapter we're going to dive deep into this topic and show you that you can do all sort of interesting things with functions in JavaScript.


I'm trying to get this image app finished. I'm working on getting the images to reblur a couple of seconds after the user clicks on them.

Alex: That sounds cool…I bet you're using `setTimeout`?

Sam: That's my plan, although I'm not sure how to know which image to reblur.

Alex: What do you mean?

Sam: I have my code so that when you click on an image and reveal it, I set up a timer that fires in two seconds. When the timeout event fires, it calls a new handler I wrote called `reblur`.

Alex: And in `reblur`, you need to know which image to reblur?

Sam: Right. I'm not passing any arguments to the handler, it's just being called by the browser when the time expires, and so I have no way to tell my handler the correct image to reblur. I'm kinda stuck.

Mara: Have you looked at the `setTimeout` documentation?

Sam: No, I only know that `setTimeout` takes a function and a time duration in milliseconds.

Mara: You can add an argument to the call to `setTimeout` that is passed on to the handler when the time event fires.

Sam: Oh, that's perfect. So I can just pass in a reference to the correct image to reblur, and that will get passed on to the handler when it's called?

Mara: You got it.

Sam: See what a little talking through code gets ya?

Alex: Oh for sure. Let's give this a try…

Finishing the image game

Now it's time to put the final polish on the image game. What we want is for an image to automatically reblur a few seconds after it's revealed. And, as we just learned, we can pass along an argument for the event handler when we call `setTimeout`. Let's check out how to do this:

```
window.onload = init;
function init() {
    let images = document.getElementsByTagName("img");
    for (let i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};

function showAnswer(eventObj) {
    let image = eventObj.target;
    let name = image.id;
    image.src = `${name}.jpg`;

    setTimeout(reblur, 2000, image);
}

function reblur(image) {
    let name = image.id;
    image.src = `${name}blur.jpg`;
}
```

This code is just as we wrote it before. No changes here...

But now when we show the user the clear image, we also call setTimeout to set up an event that will fire in two seconds.

We'll use reblur (below) as our handler and pass it 2,000 milliseconds (two seconds) and also a third argument, the image to reblur.

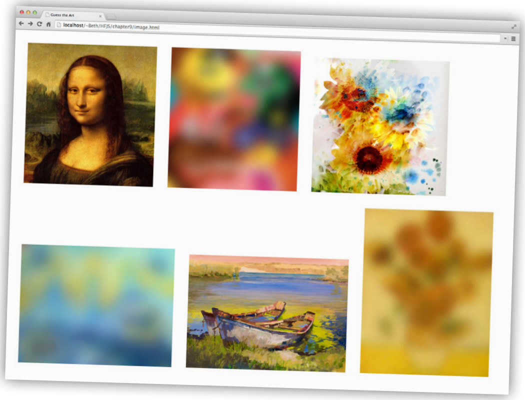
Now when this handler is called, it will be passed the image.

The handler can take the image, get the id of the image, and use that to create the name of the blurred image. When we set the src of the image to that name, it will replace the clear image with the blurred image.

Test driving the timer

That wasn't much code to add, but it sure makes a big difference in how the image game works. Now when you click, behind the scenes, the browser (through the timer events) is tracking when it needs to call the `reblur` handler, which blurs the image again. Note how *asynchronous* this feels—you're in control of when the images are clicked, but behind the scenes code is being invoked at various times based on the click event and timer events. There's no über algorithm driving things here, controlling what gets called and when; it's just a lot of little pieces of code that set up, create, and react to events.

Now when you click, you'll see the image revealed and then blurred again two seconds later.



Give this a good QA testing by clicking on lots of images in quick succession. Does it always work? Refer back to the code and wrap your brain around how the browser keeps track of all the images that need to be reblurred.

there are no Dumb Questions

Q: Can I pass just one argument to the setTimeout handler?

A: No, you can actually pass as many as you like: zero, one, or more.

Q: What about the event object? Why doesn't setTimeout pass the event handler one?

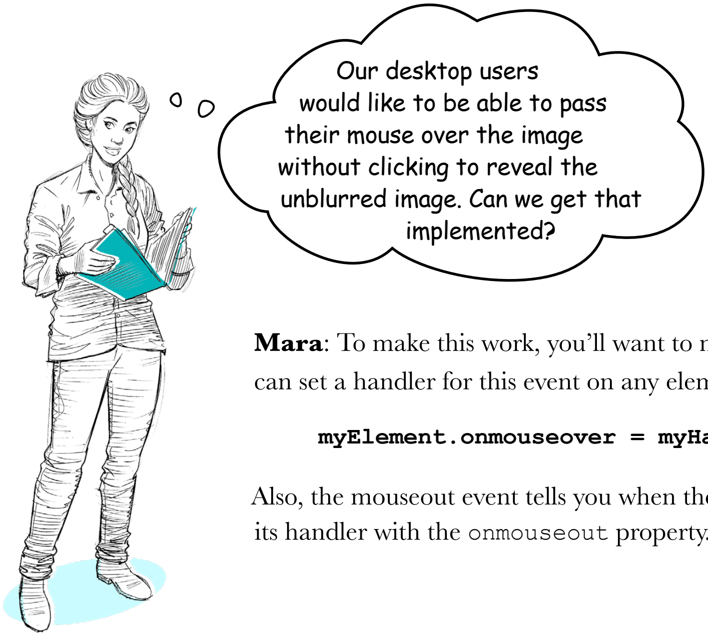
A: The event object is mostly used with DOM-related event handlers. setTimeout doesn't pass any kind of event object to its handler, because it doesn't occur on a specific element.

Q: showAnswer is a handler, and yet it sets up a new handler, reblur, in its code. Is that right?

A: You've got it. You'll actually see this fairly often in JavaScript. It's perfectly normal to see a handler set up additional event handlers for various events. This is the style of programming we referred to at the beginning of the chapter: *asynchronous programming*. To create the image game, we didn't just translate an algorithm that runs top-down. Rather, we're hooking up event handlers to handle the execution of the game as events occur. Trace through a few different examples of clicking on images and the various calls that get made to reveal and reblur the image.

Q: So there are DOM-based events, and timer events...are there lots of different kinds of events?

A: Many of the events you deal with in JavaScript are DOM events (like when you click on an element) or timer events (created with setTimeout or setInterval). There are also API-specific events, like events generated by JavaScript APIs including Geolocation, LocalStorage, Web Workers, and so on. And finally, there is a whole category of events related to I/O, like when you request data from a web service using fetch, or Web Sockets.


Our desktop users would like to be able to pass their mouse over the image without clicking to reveal the unblurred image. Can we get that implemented?

Mara: To make this work, you'll want to make use of the mouseover event. You can set a handler for this event on any element with the `onmouseover` property:

```
myElement.onmouseover = myHandler;
```

Also, the mouseout event tells you when the mouse leaves your element. You can set its handler with the `onmouseout` property.

Exercise

Rework your code so that you can reveal and reblur an image by passing your mouse over and out of the image elements. Be sure to test your code, and check your answer at the end of the chapter.

Solution on page 442

JavaScript code goes here.

Exercise

With the image game complete, Mara wrote some code to review in the weekly team meeting. In fact, she started a little contest, rewarding the first person to describe what the code does with lunch. Who wins? Sam? Alex? Or you?

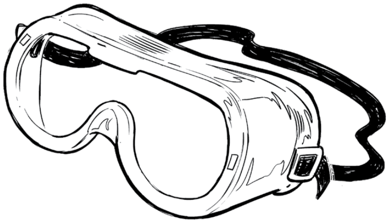
```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        let element = document.getElementById("display");
        element.innerHTML = `${element.innerHTML} that tickles!`;
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don't resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

Put your notes here describing what this code does. What events are involved? How is the handler set up? And when does the event happen? Don't just make notes, give this a try in your browser.

Solution on page 443

CODE LABORATORY

We’ve found some highly suspicious code we need your help testing. While we’ve already done an initial analysis on the code and it looks like 100% standard JavaScript, something about it seems odd. Below you’ll find two code specimens. For each specimen you’ll need to identify what appears odd about the code, test to make sure the code works, and then try to analyze what exactly it does. Go ahead and make your notes on this page. You’ll find our analysis on the next page.



Specimen #1

```
let addOne = function(x) {
                  return x + 1;
              } ;
let six = addOne(5);
```

Specimen #2

```
window.onload = function() {
    alert("The page is loaded!");
}
```

CODE LABORATORY: ANALYSIS

Specimen #1

```
let addOne = function(x) {
                return x + 1;
            } ;
let six = addOne(5);
```

At first glance, this code appears to simply define a function that adds the number one to any parameter and returns it.

Looking closer, this isn't a normal function definition. Rather, we are declaring a variable and assigning to it a function that appears to be missing its name.

Further, we're invoking the function with the variable name, not a name associated with the function as part of its definition.

Odd indeed (although it reminds us a bit of how object methods are defined).

Specimen #2

```
window.onload = function() {
    alert("The page is loaded!");
}
```

Here, we appear to have something similar. Instead of defining a function separately and assigning its name to the window.onload property, we're assigning a function directly to that property. And again, the function doesn't define its own name.

DO try this at HOME!

We added this code to an HTML page and tested it. The code appears to work as you might expect. With specimen #1, when the function assigned to addOne is invoked, we get a result that is one greater than the number we pass in, which seems right. With specimen #2, when we load the page, we get the alert "The page is loaded!"

From these tests, it would appear as if functions can be defined without names and used in places where you'd expect an expression.

What does it all mean? Stick around; we'll reveal our findings on these odd functions in the next chapter...

Bullet Points

- Most JavaScript code is written to react to **events**.
- There are many different kinds of events your code can react to.
- To react to an event, you write an **event handler** function and register it. For instance, to register a handler for the click event, you assign the handler function to the onclick property of an element.
- You're not required to handle any specific event. You choose to handle the events you're interested in.
- **Functions** are used for handlers because functions allow us to package up code to be executed later (when the event occurs).
- Code written to handle events is different from code that executes top to bottom and then completes. Event handlers can run at any time and in any order: they are **asynchronous**.
- Events that occur on elements in the DOM (DOM events) cause an **event object** to be passed to the event handler.
- The event object contains properties with extra information about the event, including the type (like "click" or "load") and the target (the object on which the event occurred).
- Many events can happen very close together. When too many events happen for the browser to handle them as they occur, the events are stored in an **event queue** (in the order in which they occurred) so the browser can execute the event handlers for each event in turn.
- If an event handler is computationally complex, it will slow down the handling of the events in the queue, because only one event handler can execute at a time.
- The functions **setTimeout** and **setInterval** are used to generate time-based events after a certain time has passed.
- The method **getElementsByTagName** returns zero, one, or more element objects in an HTMLCollection (which is array-like, so you can iterate over it).

Event Soup

click
Get this event when you click (or tap) in a web page.

load
The event you get when the browser has completed loading a web page.

mousemove
When you move your mouse over an element, you'll generate this event.

keypress
This event is generated every time you press a key.

unload
This event is generated when you close the browser window or navigate away from a web page.

mouseover
When you put your mouse over an element, you'll generate this event...

mouseout
...and you'll generate this event when you move your mouse off an element.

resize
Whenever you resize your browser window, this event is generated.

dragstart
If you drag an element in the page, you'll generate this event.

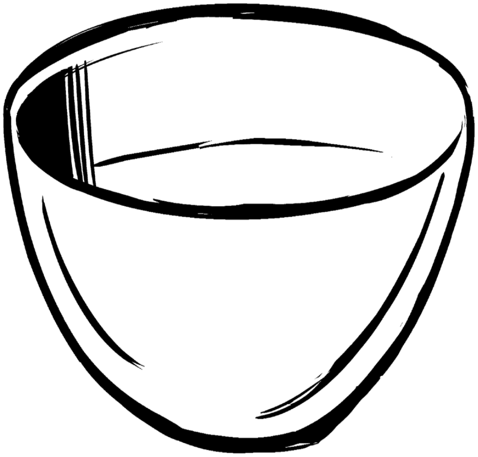
touchstart
On touch devices, you'll generate a touchstart event when you touch and hold an element...

play
Got <video> in your page? You'll get this event when you click the play button...

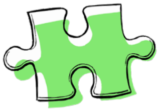
drop
You'll get this event when you drop an element you've been dragging.

pause
...and this one when you click the pause button.

touchend
...and you'll get this event when you stop touching.

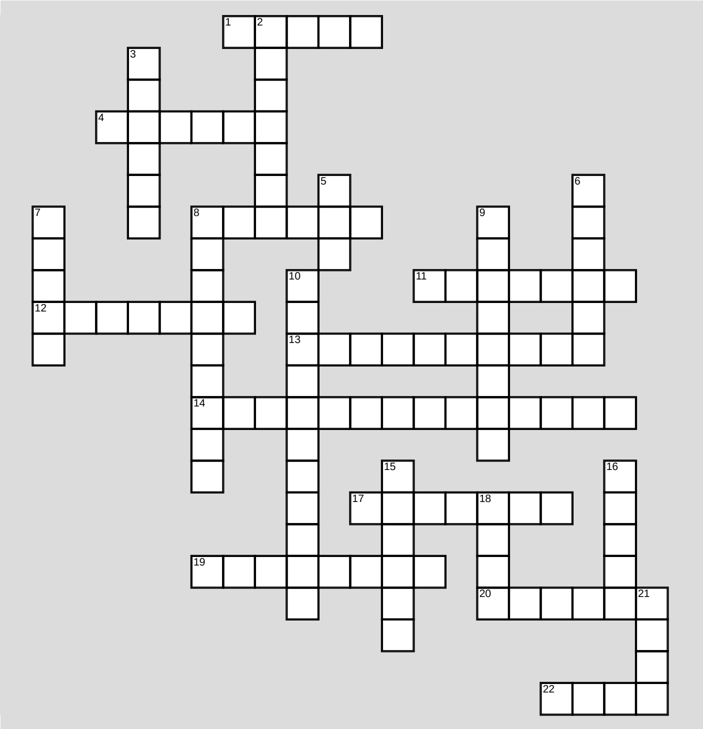


We've scratched the surface of events, using load, click, mousemove, mouseover, mouseout, resize, and timer events. Check out this delicious soup of some of the events you'll encounter in your web programming. And there's even more for you discover...



JavaScript cross

Practice your event reaction time by doing this crossword.



ACROSS

1. To pass an argument to a timer event handler, pass it as the _______ argument to setTimeout.

4. The window._______ property is for handling a page loaded event.

8. The browser has only one _______ of control.

11. A function designed to react to an event is called an event _______.

12. Events are generated for lots of things, but not for baking _______.

13. You'll generate this event if you touch your device's touch screen.

14. Events are handled _______.

17. The browser can only execute one event _______ at a time.

19. zero.jpg is the _______.

20. How you know which image was clicked on in the image game.

22. 5,000 milliseconds is _______ seconds.

DOWN

2. To assign an event handler for a time event, pass the _______ to setTimeout as the first argument.

3. A program with code to handle events is not this.

5. You can use a method to get multiple elements from the DOM using a _______ name.

6. The event _______ is passed to an event handler for DOM events.

7. When you click your mouse, you'll generate a _______ event.

8. Use this property of the event object to know when an event happened.

9. JavaScript allows you to pass a _______ to a function.

10. To make a time event happen over and over, use _______.

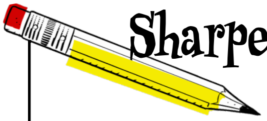
15. When you begin programming with events, you might feel like you're not in _______ anymore.

16. If too many events happen close together, the browser stores the events in an event _______.

18. The event object for a mouseover event has a property x for the position of the mouse in pixels from the _______ side of the browser window.

21. The setTimeout method is used to create a _______ event.

→ Solution on page 444



Sharpen your pencil Solution

Pick two of the events from the Brain Power on page 398. If the browser could notify your code when these events occurred, what cool or interesting code might you write? Here are a few ideas.

- ☐ Knows when the page is fully loaded and displayed
- ☐ Keeps track of all clicks made on the page, be they on a button, link, or elsewhere
- ☐ Knows when a user submits a form
- ☐ Knows when the user presses keys on the keyboard
- ☐ Knows when an element gets user interface focus
- ☐ Watches all mouse movement
- ☐ Watches the clock and manages timers and timed events
- ☐ Retrieves additional data for your page
- ☐ Tracks when the page has been resized or scrolled
- ☐ Knows when the cookies are finished baking

Let's take an event that notifies us when a user submits a form. If we're notified of this event, then we could get all the data the user filled into the form and check to make sure it's valid (e.g., that the user put something that looks like a phone number into a phone number field, or filled out the required fields). Once we've done that check, then we could submit the form to the server.

How about the mouse movement event? If we're notified whenever a user moves the mouse, then we could create a drawing application right in the browser.

If we're notified when the user scrolls down the page, we could do interesting things like reveal an image as they scroll down.

From page 408

BE the Browser Solution

Below, you'll find your game code. Your job is to play like you're the browser and figure out what you need to do after each event. Here's our solution.

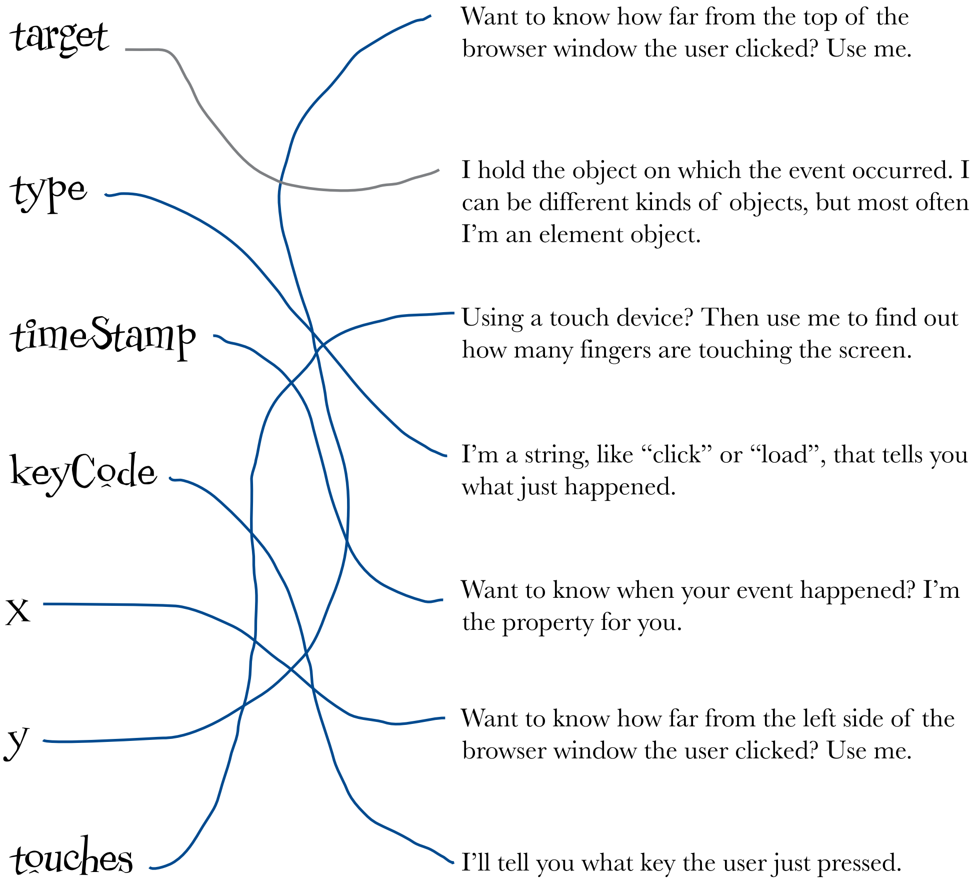
```
window.onload = init;
function init() {
    let image = document.getElementById("zero");
    image.onclick = showAnswer;
}

function showAnswer() {
    let image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

When page is being loaded...	First define the functions init and showAnswer
	Set load handler to init
When page load event occurs...	Load handler, init, is called
	Get the image with id "zero"
	Set image's click handler to showAnswer
When image click event occurs...	showAnswer is called
	Get the image with id "zero"
	Set the src attribute to "zero.jpg"

From page 416

Who Does What? Solution

You've already seen that the event object (for DOM events) has properties that give you more information about the event that just happened. Below, you'll find other properties that the event object can have. Match each event object property to what it does. Here's our solution.



target

type

timeStamp

keyCode

x

y

touches

Want to know how far from the top of the browser window the user clicked? Use me.

I hold the object on which the event occurred. I can be different kinds of objects, but most often I'm an element object.

Using a touch device? Then use me to find out how many fingers are touching the screen.

I'm a string, like "click" or "load", that tells you what just happened.

Want to know when your event happened? I'm the property for you.

Want to know how far from the left side of the browser window the user clicked? Use me.

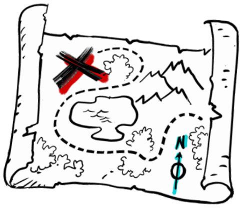
I'll tell you what key the user just pressed.

Exercise Solution

From page 421

Ahoy, matey! You've got a treasure map in your possession, and we need your help in determining the coordinates of the treasure. To do that, you're going to write a bit of code that displays the coordinates on the map as you pass the mouse over the map.

Blimey! The code is below. So far, it includes the map in the page and creates a paragraph element to display the coordinates. You need to make all the event code work. Good luck. We don't want to see you go to Davy Jones's locker anytime soon...Here's our solution.



```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Pirates Booty</title>
  <script>
      window.onload = init;
      function init() {
          let map = document.getElementById("map");
          map.onmousemove = showCoords;
      }

      function showCoords(eventObj) {
          let coords = document.getElementById("coords");
          let x = eventObj.x;
          let y = eventObj.y;
          coords.innerHTML = `Map coordinates: ${x}, ${y}`;
      }
  </script>
</head>
<body>
      <img id="map" src="map.jpg">
      <p id="coords">Move mouse to find coordinates...</p>
</body>
</html>
```

When we put our mouse right over the X, we got the coordinates: 200, 190

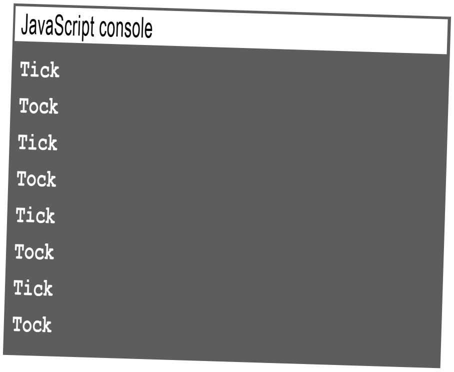
Exercise Solution

From page 426

Take a look at the code below and see if you can figure out what setInterval does. It's similar to setTimeout, but with a slight twist.Here's our solution.

Here's the code.

```
let tick = true;

function ticker() {
    if (tick) {
        console.log("Tick");
        tick = false;
    } else {
        console.log("Tock");
        tick = true;
    }
}

setInterval(ticker, 1000);
```



```
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
```

Here's the output.

Your analysis goes here.

Just like setTimeout, setInterval also takes an event handler function as its first argument and a time duration as its second argument.

But unlike setTimeout, setInterval executes the event handler multiple times...in fact, it keeps going. Forever! (Actually, you can tell it to stop; see below.) In this example, every 1,000 milliseconds (1 second), setInterval calls the ticker handler. The ticker handler checks the value of the tick variable to determine whether to display "Tick" or "Tock" in the console.

So, setInterval generates an event when the timer expires, and then restarts the timer.

```
let t = setInterval(ticker, 1000);

clearInterval(t);
```

To stop an interval timer, save the result of calling setInterval in a variable...

...and then pass that to clearInterval later, when you want to stop the timer.

Exercise Solution

From page 430

Rework your code so that you can reveal and reblur an image by passing your mouse over and out of the image elements. Be sure to test your code. Here's our solution.

```
window.onload = function() {
    let images = document.getElementsByTagName("img");
    for (let i = 0; i < images.length; i++) {
        ~~images[i].onclick = showAnswer;~~
        images[i].onmouseover = showAnswer;
        images[i].onmouseout = reblur;
    }
};
function showAnswer(eventObj) {
    let image = eventObj.target;
    let name = image.id;
    image.src = `${name}.jpg`;

    ~~setTimeout(reblur, 2000, image);~~
}

function reblur(eventObj) {
    let image = eventObj.target;
    let name = image.id;
    image.src = `${name}blur.jpg`;
}
```

First, we remove the assignment of the event handler to the onclick property.

Then, we add the showAnswer event handler to the onmouseover property of the image.

And now we're going to use reblur as the handler for the mouseout event (instead of as a timer event handler). So, we assign reblur to the onmouseout property of the image.

We won't use the timer anymore to reblur the image; instead, we'll reblur it when the user moves the mouse out of the image element.

Now we're using reblur as an event handler for the mouseout event, so to get the correct image to reblur, we have to use the event object. Just like in showAnswer, we'll use the target property to get the image object. Once we have that, the rest of reblur is the same.

Exercise Solution

From page 431

With the image game complete, Mara wrote some code to review in the weekly team meeting. In fact, she started a little contest, awarding the first person to describe what the code does with lunch. Who wins? You do! Here's our solution.

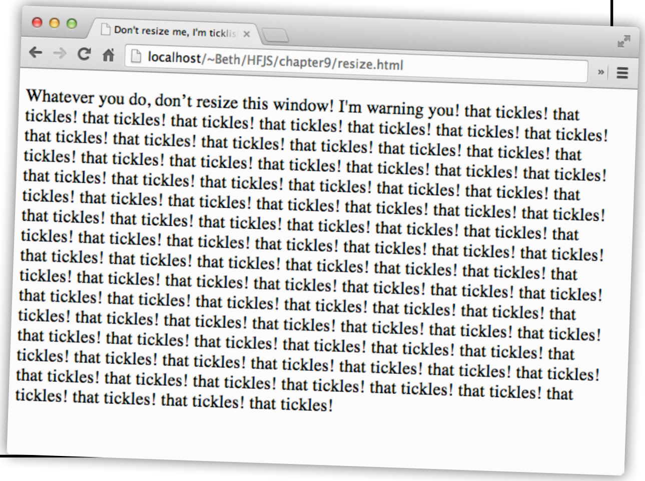
```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        let element = document.getElementById("display");
        element.innerHTML = `${element.innerHTML} that tickles!`;
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don’t resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

Our event handler is named resize. When it's called, it just adds some text to the paragraph with the id "display".

The event we're interested in is the resize event, so we set up a handler function (named resize) and assign it to the onresize property of the window.

We set up the resize event in the script at the bottom of the page. Remember, this script won't run until the page is fully loaded, so that makes sure we don't set up the event handler too early.

When you resize the browser window, the resize event handler is called, which updates the page by adding new text content (" that tickles") to the "display" paragraph.

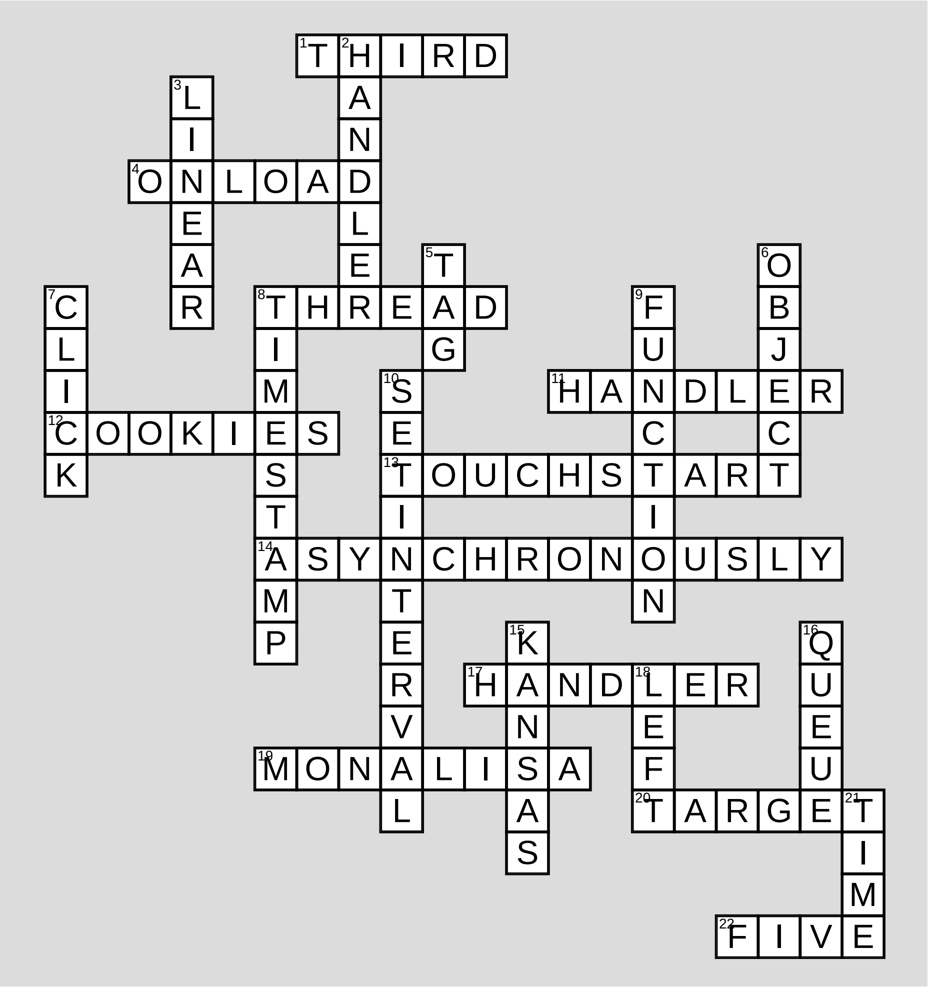

Don't resize me, I'm ticklis
localhost/~Beth/HFJS/chapter9/resize.html

Whatever you do, don't resize this window! I'm warning you! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that tickles!
that tickles! that tickles! that tickles! that tickles! that tickles! that tickles! that
tickles! that tickles! that tickles! that tickles!

JavaScript cross Solution

From page 436

Practice your event reaction time by doing this crossword. Here's our solution.


1 THIRD
2 HANDLE
3 LINEAR
4 ONLOAD
5 TAG
6 OBJECT
7 CLICK
8 THREAD
8 TIMESTAMP
9 FUNCTION
10 SETINTERVAL
11 HANDLER
12 COOKIES
13 TOUCHSTART
14 ASYNCHRONOUSLY
15 KANSAS
16 QUEUE
17 HANDLER
18 LEFT
19 MONALISA
20 TARGET
21 TIME
22 FIVE

10 anonymous and higher-order functions

Liberated functions

Know functions, then rock. Every art, craft, and discipline has a key principle that separates the intermediate players from the rock-star virtuosos—when it comes to JavaScript, it's truly understanding , then rock.**functions** that makes the difference. Functions are fundamental to JavaScript, and many of the techniques we use to **design and organize** code depend on advanced knowledge and use of functions. The path to learning functions at this level is an interesting and often mind-bending one, so get ready...The next two chapters are going to be a bit like taking Willy Wonka's tour of the chocolate factory—you're going to encounter some wild, wacky, and wonderful things as you learn more about JavaScript functions.

We'll spare you the singing Oompa Loompas.

The mysterious double life of the function keyword

So far, we've been declaring functions like this:

```
function quack(num) {
   for (let i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
```

A standard function declaration with the function keyword, a name, a parameter, and a block of code.

```
quack(3);
```

And we can invoke this function by using its name followed by parentheses that enclose any needed arguments.

There are no surprises here, but let's get our terminology down: formally, the first statement above is a *function declaration,* which creates a function that has a name—in this case, `quack`—that can be used to *reference* and *invoke* the function.

So far so good, but the story gets more mysterious because, as you saw at the end of the last chapter, there's another way to use the function keyword:

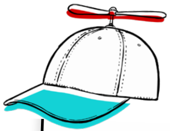
```
let fly = function(num) {
   for (let i = 0; i < num; i++) {
      console.log("Flying!");
   }
};
```

This doesn't look so standard: the function doesn't have a name, and it's on the right-hand side of an assignment to a variable.

```
fly(3);
```

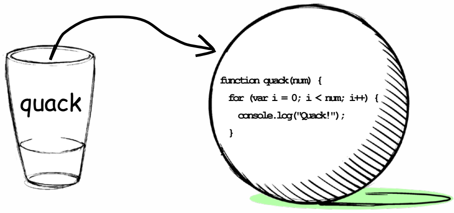
We can invoke this function too, this time by using the variable fly.

Now, when we use the function keyword this way—that is, within a statement, like an assignment statement—we call this a *function expression*. Notice that, unlike the function declaration, this function doesn't have a name. Also, the expression results in a value that is then assigned to the variable `fly`. What is that value? Well, we're assigning it to the variable `fly` and then later invoking it, so it must be a *reference to a function*.



Serious Coding

A function reference is exactly what it sounds like: a reference that refers to a function. You can use a function reference to invoke a function or, as you'll see, you can assign them to variables, store them in objects, and pass them to or return them from functions (just like object references).



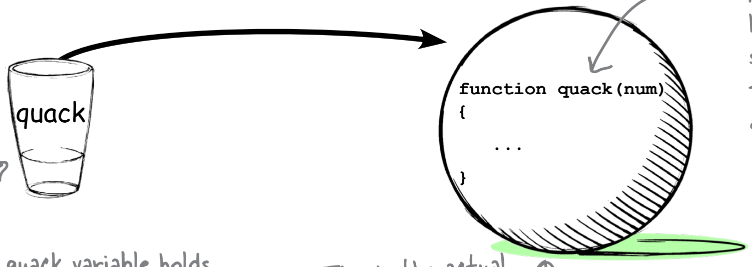
```
function quack(num) {
  for (var i = 0; i < num; i++) {
    console.log("Quack!");
  }
```

How functions are values too

Sure, we all think of functions as things we invoke, but you can think of functions as *values* too. That value is actually a reference to the function, and as you've seen, whether you create a function with a function declaration or a function expression, you get a reference to that function:

```
function quack(num) {
    ...
}
```

This code creates a variable and a function, like this:



Notice the quack function has a name, quack—the same name as the variable the function reference is assigned to.

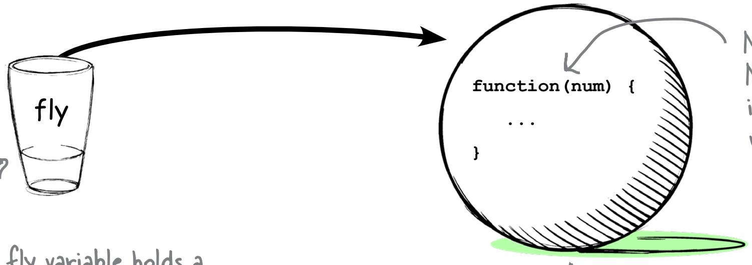
The quack variable holds a reference to a function.

This is the actual quack function.

Whether we declare a function, like with quack, or assign a function expression to a variable, like with fly, we get a variable that holds a reference to a function.

```
let fly = function(num) {
    ...
};
```

This code creates a variable and a function, like this:



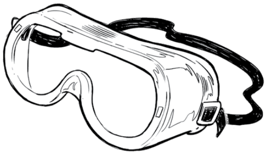
Notice the fly function does NOT have a name. We refer to it using the variable the function reference is assigned to.

The fly variable holds a reference to a function.

This is the actual fly function.

Start thinking about functions as values, just like numbers, strings, booleans, or objects. The thing that really makes a function value different from these other values is that we can invoke it.

IN THE LABORATORY



Time to get back into the laboratory to look under the covers of function declarations and function expressions. Let's use the **typeof** operator again on our two functions, quack and fly.

Collect the data for the following experiments:

Here's the test data...

```
function quack(num) {
   for (let i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
let fly = function(num) {
   for (let i = 0; i < num; i++) {
      console.log("Flying!");
   }
};
console.log(typeof quack);
console.log(typeof fly);
```

...and the tests.

JavaScript console

Put your results here. Are there any surprises?

there are no Dumb Questions

Q: We've seen expressions like 3+4 and Math.random() * 6, but how can a function be an expression?

A: An expression is anything that evaluates to a value. 3+4 evaluates to 7, Math.random() * 6 evaluates to a random number, and a function expression evaluates to a function reference.

Q: But a function declaration is not an expression?

A: No, a function declaration is a statement. Think of it as having a hidden assignment that assigns the function reference to a variable for you. A function expression doesn't assign a function reference to anything; you have to do that yourself.

Q: What good does it do me to have a variable that refers to a function?

A: Well, for one thing, you can use it to invoke the function:

```
myFunctionReference();
```

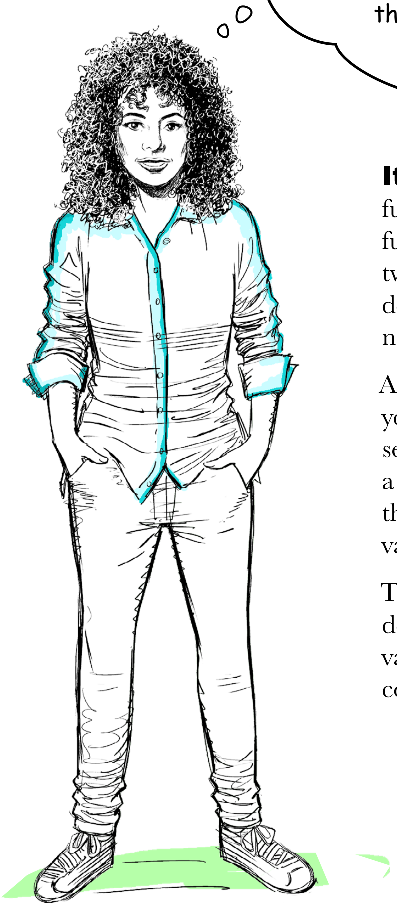
You can also pass a reference to a function or return a reference from a function, but we're getting a little ahead of ourselves. We'll come back to this in a few pages.

Q: Can function expressions only appear on the right-hand side of an assignment statement?

A: Not at all. A function expression can appear in many different places, just like other kind of expressions can. Stay tuned, because this is a really good question and we'll be coming back to this in just a bit.

Q: Okay, a variable can hold a reference to a function. But what is the variable really referencing? Just some code that is in the body of the function?

A: That's a good way to begin thinking about functions, but think of them more as a little crystallized version of the code, all ready to pull out at any time and invoke. You're going to see later that this crystallized function has a bit more in it than just the code from the body.

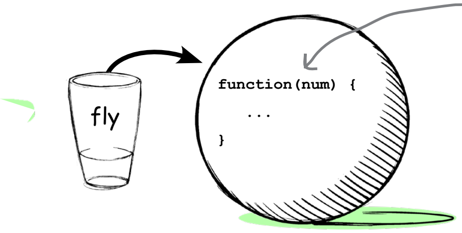

I'm not seeing much of a difference here between quack and fly. They are both functions. They both create a function reference that's stored in a variable. We invoke them both in exactly the same way. I feel like I'm missing something subtle here.

It is subtle. First of all, you're right—whether you use a function declaration or a function expression, you end up with a function. But there are some important differences between the two. For one, function declarations and function expressions are defined at different times. We'll talk about this some more in the next chapter.

Another difference has to do with function naming—when you use a declaration, the function name is used to create and set up a variable that refers to the function. And when you use a function expression, you typically don't provide a name for the function, so either you end up assigning the function to a variable in code, or you use the function expression in other ways.

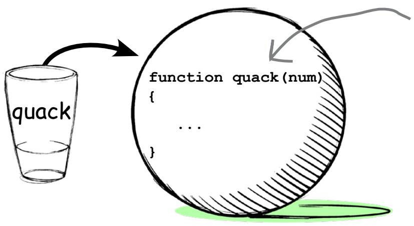
We'll take a look at what those are later in this chapter.

The key here is to think of the results of both function declarations and function expressions as *values*, just like the other values we deal with in JavaScript programming...except, of course, functions can be invoked (or called).


fly
function(num) {
...
}

The function fly has no name.

For now, just remember how function declarations and function expressions are evaluated and how the names are handled. All of this will become useful shortly.


quack
function quack(num)
{
...
}

The function quack has a name.

In all other respects, they are the same: they are both function values. Both fly and quack are assigned to variables, and both function values can be invoked.

If functions are values, we can assign them to variables

So far, we've seen two ways to create functions—the function declaration (a statement) and the function expression (an expression):

```
function quack(num) {
   for (let i = 0; i < num; i++) {
      console.log("Quack!");
   }
}

let fly = function(num) {
   for (let i = 0; i < num; i++) {
      console.log("Flying!");
   }
}
```

Our two functions again. Remember, quack is defined with a function declaration, and fly with a function expression. Both result in function references, which are stored in the variables quack and fly, respectively.

The function declaration takes care of assigning the reference to a variable with the name you supply, in this case quack.

When you have a function expression, you need to assign the resulting reference to a variable yourself. Here, we're storing the reference in the fly variable.

Once we have defined a function—that is, created a function value—we can do things with that value. For instance, we can assign the values to other variables, like this:

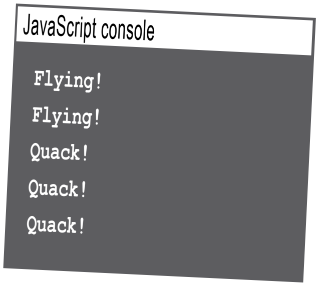
```
let superFly = fly;
superFly(2);
```

After we assign the value in fly to superFly, superFly holds the function reference, so by adding some parentheses and an argument we can invoke it!

```
let superQuack = quack;
superQuack(3);
```

And even though quack was created by a function declaration, the value in quack is a function reference too, so we can assign it to the variable superQuack and invoke it.

In other words, references are references, no matter how you create them (that is, with a declaration or an expression)!



JavaScript console

```
Flying!
Flying!
Quack!
Quack!
Quack!
```

Sharpen your pencil

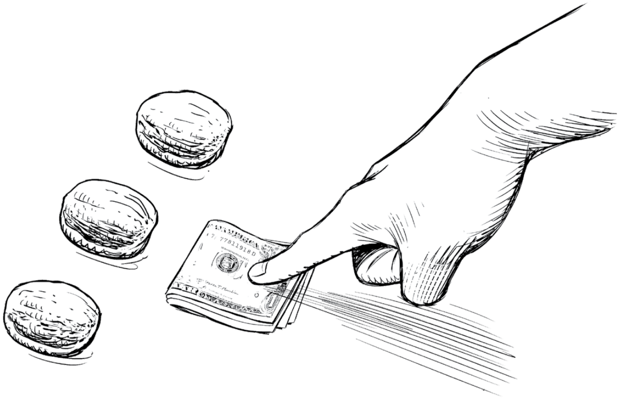
To get the idea of functions as values into your brain, let's play a little game of chance. Try the shell game. Will you win or lose? Give it a try and find out.

```
let winner = function() { alert("WINNER!") };
let loser = function() { alert("LOSER!") };
// let's test as a warm-up
winner();
// let's assign to other variables for practice
let a = winner;
let b = loser;
let c = loser;
a();
b();
// now let's try your luck with a shell game
c = a;
a = b;
b = c;
c = a;
a = c;
a = b;
b = c;
a();
```

Remember, these variables hold references to the winner and loser functions. We can assign and reassign these references to other variables, just like with any value.

And at any time, we can invoke a reference to a function.

Execute the code (by hand!) and figure out if you won or lost.



Are you thinking about functions as values now?

Solution on page 484

Function Exposed

This week's interview: Understanding Function

Head First: Function, we're so happy to finally have you back on this show. You're quite a mystery, and our readers are dying to know more.

Function: It's true, I'm deep.

Head First: Let's start with this idea that you can be created with a declaration, or created with an expression. Why two ways of defining you? Wouldn't one be enough?

Function: Well, remember, these two ways of creating a function do two slightly different things.

Head First: But the result is the same: a function, right?

Function: Yes, but look at it this way. A function declaration is doing a little bit of work behind the scenes for you: it's creating the function, and then also creating a variable to store the function reference in. A function expression creates the function, which results in a reference, and it's up to you to do something with it.

Head First: But don't we always just store the reference we get from a function expression in a variable anyway?

Function: Definitely not. In fact, we usually don't. Remember, a function reference is a value. Think of the kinds of things you can do with other kinds of values, like an object reference, for instance. I can do all those things too.

Head First: But how can you possibly do everything those other values can do? I declare a function, and I call it. That's about as much as any language allows, right?

Function: Wrong. You need to start thinking of a function as a value, just like objects or the primitive types. Once you get hold of a function, you can do all kinds of things with it. But there is one important difference between a function and other kinds of values, and that is what really makes me what I am: a function can be invoked, to execute the code in its body.

Head First: That sounds very impressive and powerful, but I'd have no idea what to do with you other than define and call you.

Function: This is where we separate the six-figure coders from the scripters. When you can treat a function like any other value, all kinds of interesting programming constructs become possible.

Head First: Can you give us just one example?

Function: Sure. Say you want to write a function that can sort *anything*. No problem. All you need is a function that takes two things: the collection of items you need to sort, and *another function* that knows how to compare any two items in your collection. Using JavaScript, you can easily create code like that. You write one sort function for every kind of collection, and then just tell that function how to compare items by passing it a function that knows how to do the comparison.

Head First: Err…

Function: Like I said, this is where we separate the six-figure coders from the scripters. Again, we're *passing a function* that knows how to do the comparison *to the other function*. In other words, we're treating the function like a value by passing it to a function *as a value*.

Head First: And what does that get us other than confused?

Function: It gets you less code, less hard work, more reliability, better flexibility, better maintainability, a higher salary...

Head First: That all sounds good, but I'm still not sure how to get there.

Function: Getting there takes a little bit of work. This is definitely an area where your brain has to expand a bit.

Head First: Well, Function, my head is expanding so much it's about to explode, so I'm going to go lie down.

Function: Oh, and by the way, there's a third way you can define me, not just two...

Head First: Errr...now I *really* need to lie down.

Did we mention functions have first class status in JavaScript?

If you're coming to JavaScript from a more traditional programming language you might expect functions to be…well, just functions. You can declare them and call them, but when you're not doing either of those things they just sit around doing nothing.

Now, you know that functions in JavaScript are values—values that can be assigned to variables. And you know that with values of other types, like numbers, booleans, strings, and even objects, we can do all sorts of things with those values, like pass them to functions, return them from functions, or even store them in objects or arrays.

Computer scientists actually have a term for these kinds of values: they're called *first class values*. Here's what you can do with a first class value:

☐ Assign the value to a variable (or store it in a data structure like an array or object).

☐ Pass the value to a function.

☐ Return the value from a function.

Guess what? We can do all these things with functions too. In fact, we can do *everything* with a function that we can do with other values in JavaScript. So consider functions first class values in JavaScript, along with all the values of the types you already know: numbers, strings, booleans, and objects.

Here's a more formal definition of first class:

> **First class**: A value that can be treated like any other value in a programming language, including the ability to be assigned to a variable, passed as an argument, and returned from a function

We're going to see that JavaScript functions easily qualify as first class values—in fact, let's spend a little time working through just what it means for a function to be first class in each of these cases. Here's a little advice: stop thinking about functions as something special and different from other values in JavaScript. There is great power in treating a function like a value. Our goal in the rest of the chapter is to show you why.

Taking a look at the other side of functions...

You've already seen two sides of functions—you've seen the formal, declarative side of function declarations, and you've seen the looser, more expressive side of function expressions. Well, now it's time to introduce you to another interesting side of functions: *the anonymous side.*

By anonymous, we're referring to functions *that don't have names.* How can that happen? Well, when you define a function with a function declaration, your function will *definitely have a name*. But when you define a function using a function expression, *you don't have to give that function a name*.

You're probably saying, sure, that's an interesting fact, maybe it's possible, but so what? By using anonymous functions, we can often make our code less verbose, more concise, more readable, more efficient, and even more maintainable.

So, let's see how to create and use anonymous functions. We'll start with a piece of code we've seen before, and see how an anonymous function might help out:

Here's a load handler, set up like we've always done in the past.

First, we define a function. This function has a name, handler.

```
function handler() { alert("Yeah, that page loaded!"); }
window.onload = handler;
```

Then we assign the function to the onload property of the window object, using its name, handler.

And when the page loads, the handler function is invoked.

Sharpen your pencil

Use your knowledge of functions and variables and check off the true statements below.

- ☐ The handler variable holds a function reference.
- ☐ When we assign handler to window.onload, we're assigning it a function reference.
- ☐ The only reason the handler variable exists is to assign it to window.onload.
- ☐ We'll never use handler again, as it's code that is meant to run only when the page first loads.
- ☐ Invoking onload handlers twice is not a great idea—doing so could cause issues given that these handlers usually do some initialization for the entire page.
- ☐ Function expressions create function references.
- ☐ Did we mention that when we assign handler to window.onload, we're assigning it a function reference?

→ Solution on page 485

How to use an anonymous function

So, we're creating a function to handle the load event, but we know it's a "one-time" function because the load event only happens once per page load. We can also observe that the `window.onload` property is being assigned a function reference—namely, the function reference in `handler`. But because `handler` is a one-time function, that name is a bit of a waste, because all we do is assign the reference in it to the `window.onload` property.

Anonymous functions give us a way to clean up this code. An anonymous function is just a function expression without a name that's used where we'd normally use a function reference. But to make the connection, it helps to see how we use a function expression in code in an anonymous way:

```
function ~~handler~~() { alert("Yeah, that page loaded!"); }
window.onload = ~~handler~~;
```

First, remove the handler variable so this becomes a function expression.

```
         function() { alert("Yeah, that page loaded!"); }
window.onload =                                          ;
```

Then, assign the function expression directly to the window.onload property.

```
window.onload = function() { alert("Yeah, that page loaded!"); };
```

Now the handler has been assigned to the window.onload property without the need for an unnecessary name.

Much more concise now. We're assigning the function we need directly to the onload property. We also aren't creating a function name that might mistakenly be used in other code (after all, handler is a pretty common name!).

Look, Ma! No names!

Brain Power

Are there places in your previous code that you've seen anonymous functions and hadn't realized it?

Hint: are they hiding somewhere in your objects?

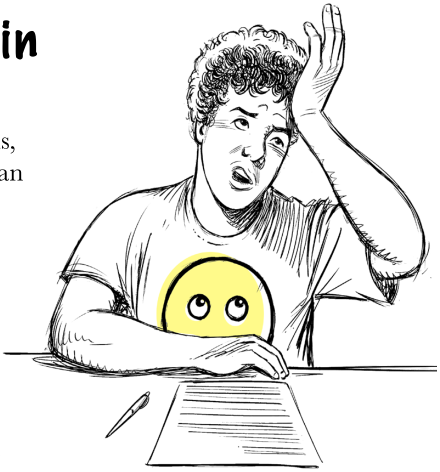
Sharpen your pencil

There are a few opportunities in the code below to take advantage of anonymous functions. Go ahead and rework the code to use anonymous functions wherever possible. You can scratch out the old code and write in new code where needed. Oh, and one more task: circle any anonymous functions that are already being used in the code.

```
window.onload = init;
let cookies = {
    instructions: "Preheat oven to 350...",
    bake(time) {
        console.log("Baking the cookies.");
        setTimeout(done, time);
    }
};
function init() {
    let button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    let cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

→ Solution on page 486

We need to talk about your verbosity, again



Okay, we hate to bring it up again because you've come a long way with functions—you know how to assign functions to variables, pass them to functions, return them from functions—but, well, you're still being a little more verbose than you have to (we could also say you're not being as expressive as you could be). Let's see an example:

Here's a normal-looking function named cookieAlarm that displays an alert about cookies being done.

```
function cookieAlarm() {
    alert("Time to take the cookies out of the oven");
}

setTimeout(cookieAlarm, 600000);
```

Looks like the cookies will be done in 10 minutes; just sayin'.

In case you forgot, these are milliseconds, so 1,000 * 60 * 10 = 600,000.

And here we're taking the function and passing it as an argument to setTimeout.

While this code looks fine, we can make it a bit tighter using anonymous functions. How? Well, think about the `cookieAlarm` variable in the call to `setTimeout`. This is a variable that references a function, so when we invoke `setTimeout`, a function reference is passed. Now, using a variable that references a function is one way to get a function reference, but just like with the `window.onload` example a couple of pages back, you can use a function expression too. Let's rewrite the code with a function expression instead:

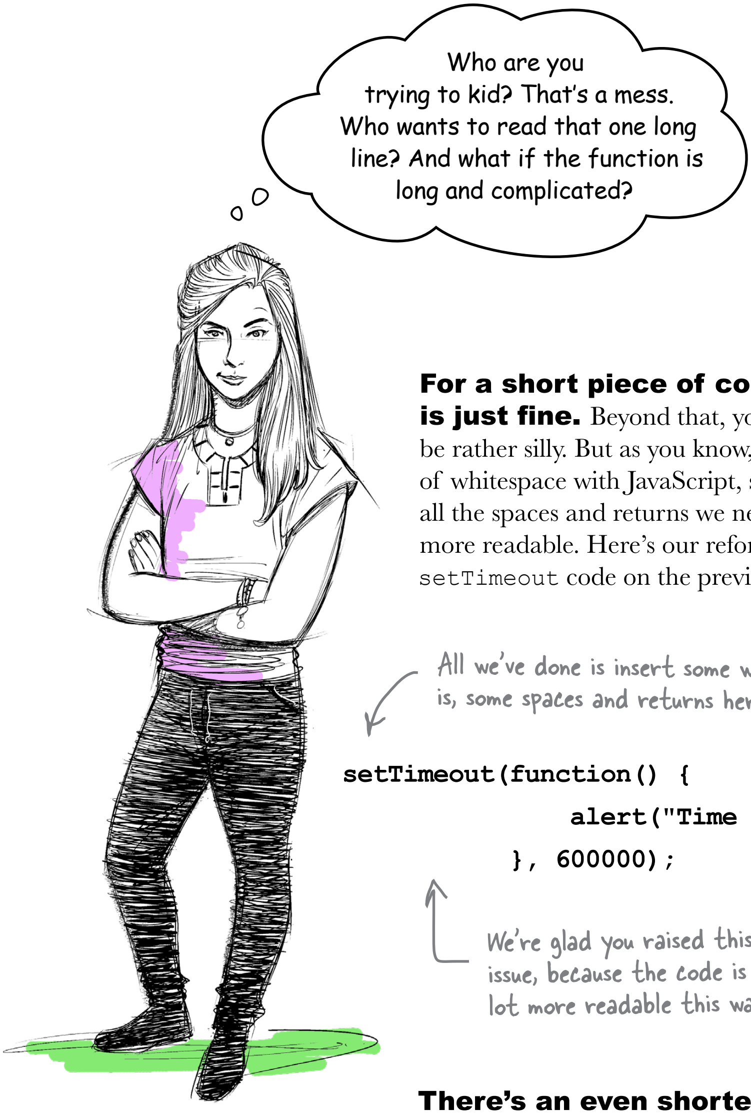
Now instead of a variable, we're just putting the function, inline, in the call to setTimeout.

Pay careful attention to the syntax here. We write the entire function expression, which ends in a right bracket, and then, like any argument, we follow it with a comma before adding the next argument.

```
setTimeout(function() { alert("Time to take the cookies out of the oven");}, 600000);
```

We specify the name of the function we're calling, setTimeout, followed by a parenthesis and then the first argument, a function expression.

Here's the second argument, after the function expression.


Who are you trying to kid? That's a mess. Who wants to read that one long line? And what if the function is long and complicated?

For a short piece of code, a one-liner is just fine. Beyond that, you're right, it would be rather silly. But as you know, we can use lots of whitespace with JavaScript, so we can insert all the spaces and returns we need to make things more readable. Here's our reformatting of the `setTimeout` code on the previous page:

All we've done is insert some whitespace...that is, some spaces and returns here and there.

```
setTimeout(function() {
          alert("Time to take the cookies out of the oven");
       }, 600000);
```

We're glad you raised this issue, because the code is a lot more readable this way.

There's an even shorter way to write this code. We're not done yet. We can make the cookie alarm function even more concise by using a third way to define a function: the *arrow function*. What's an arrow function? Check this out:

```
setTimeout(() => alert("Time to take the cookies out of the oven"), 600000);
```

This is an arrow function.

We can make the code even shorter with arrow functions

So far in this book, we've focused on two different ways to make functions: *function declarations* and *function expressions*. There's a third way to make functions, and that's the *arrow function*. Arrow functions are almost the same thing as anonymous function expressions (except for a couple of important differences we'll get to shortly), and they can make your code even more concise.

Let's take a look at how you make an arrow function. We've already stepped you through how to convert a function declaration into an anonymous function expression like this:

```
function ~~handler~~() { alert("Yeah, that page loaded!"); }
window.onload = ~~handler~~;
```

```
window.onload = function() { alert("Yeah, that page loaded!"); };
```

We can convert the handler function into an arrow function like this:

Step 1: Get rid of the function keyword:

```
window.onload = ~~function~~() { alert("Yeah, that page loaded!"); };
```

Step 2: Add the arrow:

This is the arrow in an arrow function.

```
window.onload = () => { alert("Yeah, that page loaded!"); };
```

Step 3: Get rid of the { and } (you can do this only if there's just one statement in the function):

```
window.onload = () => ~~{~~ alert("Yeah, that page loaded!"); ~~}~~;
```

Step 4: Finally, get rid of the ; (again, you can do this only if there's just one statement in the function):

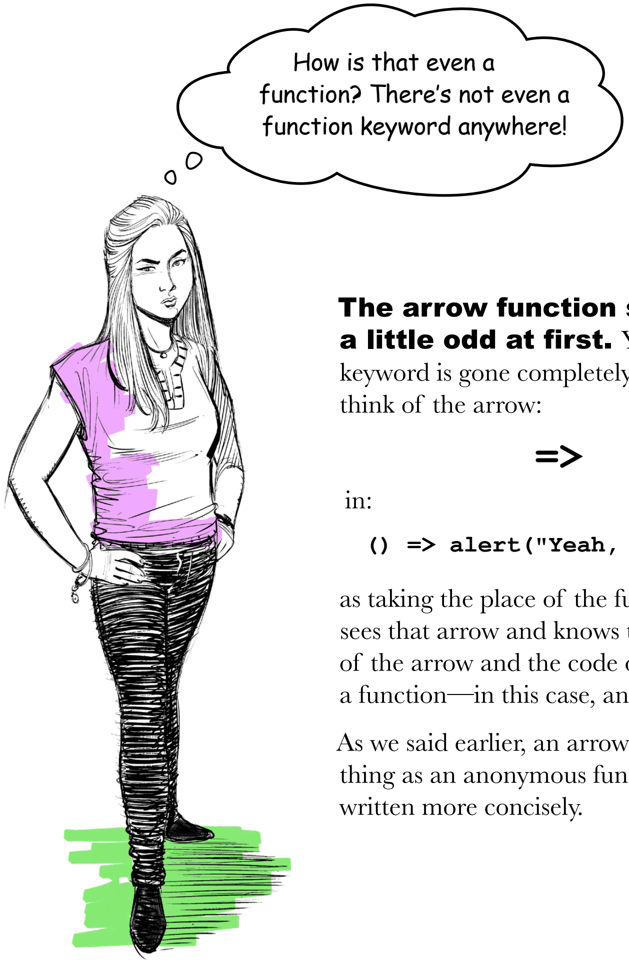
```
window.onload = () => alert("Yeah, that page loaded!")~~;~~;
```

Make sure you remove only the semicolon that is on the alert statement, not the one on the assignment.

The final result is:

```
window.onload = () => alert("Yeah, that page loaded!");
```

Shorter!


How is that even a function? There's not even a function keyword anywhere!

The arrow function syntax might look a little odd at first. You're right, the function keyword is gone completely in arrow functions. You can think of the arrow:

=>

in:

```
() => alert("Yeah, that page loaded!")
```

as taking the place of the function keyword. The browser sees that arrow and knows that the parentheses on the left of the arrow and the code on the right of the arrow define a function—in this case, an arrow function.

As we said earlier, an arrow function is almost the same thing as an anonymous function expression; it's just written more concisely.

Sharpen your pencil

Rewrite the cookie alarm function as an arrow function in the space below. Check your answer at the end of the chapter.

Current version of the function.

```
setTimeout(function() { alert("Time to take the cookies out of the oven");}, 600000);
```

Write the equivalent arrow function here.

```
setTimeout( ________________________________________________ , 600000);
```

Solution on page 485

Creating arrow functions

Arrow functions are particularly useful when you've got short, one-line, anonymous functions with no or just a few parameters, and you only need those functions once. A window `onload` or `setTimeout` handler can be a great place to use an arrow function. Typically, these are short, one-use functions, so there's no need to give them a name, and you want to write them as concisely as you can.

Let's make an arrow function out of a simple function that takes a number, adds one to it, and returns that value. We'll start by writing that function as an anonymous function expression, like this:

```
let addOne = function(n) { return n + 1; };
```

Now, using the rules we already talked about, we can convert this to an arrow function by removing the `function` keyword, { and }, and semicolon after the `return` and adding an arrow, like this:

let addOne = ~~function~~(n) => ~~{~~ return n + 1~~;~~ ~~}~~;

```
let addOne = (n) => return n + 1;
```

However, if you try this code, you'll get an error. Why?

There's one more "rule of conciseness" you need to apply in this arrow function: when the only statement in an arrow function is a return statement, you must eliminate the `return`. The value of the expression that's left is automatically returned by the function (instead of undefined). If we apply that rule, we get:

```
let addOne = (n) => n + 1;
```

If you really want to keep the `return`, or you have a function with multiple statements in the body of the function (or both), you must use the { and } around the function body. For example, let's rewrite `addOne` as an arrow function with multiple statements, including a return statement:

```
let addOne = (n) => {
    let m = n + 1;
    return m;
};
```

Here we have multiple statements in the body of the function AND we have a return statement, so we must use { and } to delimit the body of the function.

Note, too, that we can write an arrow function on multiple lines, just like we can anonymous function expressions.

Lastly, before we look at how we might use arrow functions in our code, how about a function with two parameters? We can do that like this:

```
let addTwoNums = (n, m) => n + m;
```

Just like in a regular function, we put multiple parameters in ( ), separated by a comma.

there are no Dumb Questions

Q: Using these anonymous functions like this seems really esoteric. Do I really need to know this stuff?

A: You do. Anonymous function expressions are used frequently in JavaScript code, so if you want to be able to read other people's code, or understand JavaScript libraries, you're going to need to know how these work, and how to recognize them when they're being used.

Q: Is using an anonymous function expression really better? I think it just complicates the code and makes the code hard to follow and read.

A: Give it some time. With practice, you'll be able to parse code like this more easily when you see it, and there really are lots of cases where this syntax decreases code complexity, makes the code's intention clearer, and cleans up your code. That said, overuse of this technique can definitely lead to code that is quite hard to understand—but stick with it, and it'll get easier to read and more useful as you get the hang of it. You're going to encounter lots of code that makes heavy use of anonymous functions, so it's a good idea to incorporate this technique into your code toolbelt.

Q: If first class functions are so useful, how come other languages don't have them?

A: Ah, but they do (and even the ones that don't are considering adding them). For instance, languages like Scheme and Scala have fully first class functions like JavaScript does. Other languages, like PHP, Java (in the newest versions), C#, and Objective C have some or most of the first class features that JavaScript does. As more people are recognizing the value of having first class functions in a programming language, more languages are supporting them. Each language does it a little differently, however, so be prepared for a variety of approaches as you explore this topic in other languages.

Q: The arrow in the arrow function looks a lot like the "greater than or equals" operator; can it also mean that sometimes, in the right context?

A: No. >= is a always a conditional operator and => is always indicating an arrow function. If you try to write 3 => 4, you'll get an error. Try it in the console!

Q: You said an arrow function is slightly different from a function expression—how?

A: Arrow functions are intended to be a more concise alternative to function expressions, as we've seen. The main goal is conciseness. However, arrow functions have slightly different semantics from function expressions. For instance, they shouldn't be used as methods, because **this** is not assigned the value of the object whose method you're calling (like it is in regular object methods). There are some other subtle differences, but that's the main one.

Q: How do I know if I should use a function expression or an arrow function?

A: If you're passing a one-off function to another function, that's a great place to use an arrow function. If, however, you're writing a method of an object, or a handler for an event, like onload or onclick, use a function expression instead, because in those cases, you'll probably need **this** to be assigned a reference to the object whose method you're calling, or the element object associated with the event handler.

We'll see quite a few places later in the chapter where using an arrow function is concise and a great solution, so there'll be lots more examples coming up.

Anonymous and arrow functions seem to work well with the setTimeout function. What else can they do?

Anonymous functions, and their arrow function shorthand, open up all kinds of new possibilities.
We've already seen one example of this, when we used functions with events in the last chapter. If you think through the functions we supplied to our events, they can all be specified as anonymous functions. Anonymous and first-class functions are also going to allow us to parameterize other functions with computation in something called *higher-order* functions. We'll get to that shortly and see its power. We'll even go further in the next chapter and explore how these functions enable us to use a powerful technique called *closures* to enable keeping private state in our functions. And ahead in your studies, you'll even discover functions as the foundation of how JavaScript handles asynchronous computation.

That's a lot! We'd better get started.

Webville Cola

Webville Cola needs a little help managing the code for its product line. To give them a hand, let's take a look at the data structure they use to hold the sodas they produce:

Looks like they're storing their products as an array of objects. Each object is a product.

```
let products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];
```

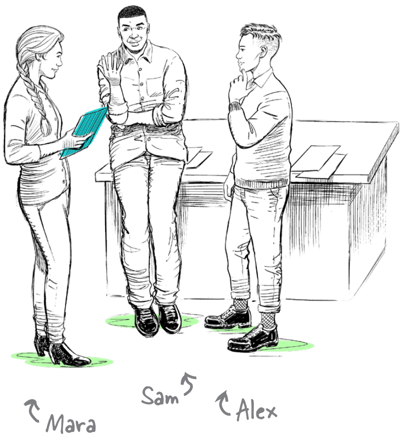
In each product they're storing a name, number of calories, color, and number of bottles sold per month.

They really need some help sorting these products. Sometimes they need to sort them by bottles sold, sometimes by calories, sometimes by name.

Brain Power

Why do we have function parameters? What effect do they have on a function?

(This question might seem odd right now, but think about it, and you'll see why we're asking it shortly.)



Sam: I got a call from Webville Cola, and they need help with their product data. They want to be able to sort their products by any property, like name, bottles sold, soda color, calories per bottle, and so on, but they want it flexible in case they add more properties in the future.

Alex: How are they storing this data?

Sam: Oh, each soda is an object in an array, with properties for name, number sold, calories…

Alex: Got it.

Sam: My first thought was just to look for a simple sort algorithm and implement it. Webville Cola doesn't have many products, so it can be simple.

Mara: Oh, I've got a simpler way than that, but it requires you to use your knowledge of first class functions.

Sam: I like hearing simple! But how do first class functions fit in? That sounds complicated to me.

Mara: Not at all. It's as easy as writing a function that knows how to compare two values, and then passing that to another function that does the real sorting for you.

Alex: How does the function we're writing work, exactly?

Mara: Well, instead of handling the entire sort, all you need to do is write a function that knows how to compare two values. Say you want to sort by a product property like the number of bottles sold. You set up a function like this:

```
function compareSold(product1, product2) {
  // code to compare here
}
```

This function needs to take two products and then compare them.

We can fill in the details of the code in a minute, but for now, the key is that once you have this function, you just pass it to a sort function and that sort function does the work for you—it just needs you to help it know how to compare things.

Sam: Wait, where is this sort function?

Mara: It's actually a method that you can call on any array. So you can call the `sort` method on the `products` array and pass it this compare function we're going to write. And, when `sort` is done, the `products` array will be sorted by whatever criteria `compareSold` used to sort the values.

Alex: So if I'm sorting how many bottles are sold, those are numbers, so the `compareSold` function just needs to determine which value is less or greater?

Mara: Right. We'd better take a closer look at how the array sort works…

Understanding the array sort method

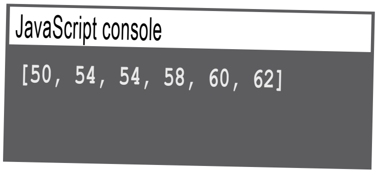
JavaScript arrays provide a `sort` method that, given a function that knows how to compare two items in the array, sorts the array for you. There's a lot in that sentence, so let's break it down with a simple example. Here's a simple array of numbers:

```
let numbersArray = [60, 50, 62, 58, 54, 54];
```

Just an ordinary array filled with some numbers that we'd like to sort.

Now, there are all kinds of ways we could sort these numbers, but let's say we want to sort them in descending order from large to small. If we use the array's `sort` method with no argument at all, that's not what we get:

```
let numbersArray = [60, 50, 62, 58, 54, 54];
numbersArray.sort();
console.log(numbersArray);
```



```
[50, 54, 54, 58, 60, 62]
```

Here we're using the array's sort method, but it sorts ascending by default. Hmm...

Notice that the sort method sorts the original array, as opposed to, say, creating a new one.

The array's `sort` method can also take a single argument, a comparison function. Here's how that works: the comparison function takes two items, and compares them. It returns −1 if the first item is less than the second, 0 if they are equal, or 1 if the second item is greater than the first. Let's see if we can write a comparison function to compare numbers:

```
function compare(num1, num2) {
    if (num1 < num2) {
        return 1;
    } else if (num1 === num2) {
        return 0;
    } else {
        return -1;
    }
}
```

Our function is always passed two values to compare.

If the first is less than the second we return 1.

If the two are equal, we return 0.

And the only remaining case is if the first is larger than the second, then we return −1.

Putting it all together

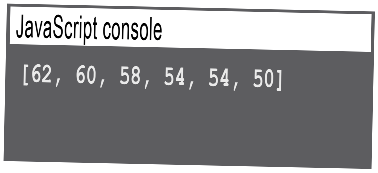
Now that we've written a `compare` function, all we need to do is call the `sort` method on `numbersArray` and pass it the function. Here's how we do that:

```
let numbersArray = [60, 50, 62, 58, 54, 54];

numbersArray.sort(compare);

console.log(numbersArray);
```

We call the sort method on the array, passing it the compare function.

When sort is complete, the array is sorted in descending order, and we display that in the console as a sanity check.

Here's the array sorted in descending order.



```
[62, 60, 58, 54, 54, 50]
```

Note again that the sort method is destructive; that is, it changes the array, rather than returning a new array that is sorted.

Sharpen your pencil

You know that the comparison function we pass to sort needs to return a number greater than 0, equal to 0, or less than 0, depending on the two items we're comparing. To get a descending-ordered sort, here's what we do: if the first item is less than the second, we return a value greater than 0; if the first item is equal to the second, we return 0; and if the first item is greater than the second, we return a value less than 0.

Can you use this knowledge with the fact that we're comparing two numbers in compare to rewrite compare using much less code?

Check your answer at the end of the chapter before you go on.

Hint: you won't need this much space.

Another hint: if you are comparing two numbers, what do you get if you subtract one from the other?

→ Solution on page 485

Meanwhile, back at Webville Cola

Let's get back to helping Webville Cola, with our new knowledge of array sorting. Of course, all we really need to do is write a comparison function for them, but before we do that let's quickly review the `products` array again:

But we don't have to tell them that.

```
let products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];
```

Remember, each item in the products array is an object. We don't want to compare the object references to one another; we want to compare specific properties, like sold, in the objects.

So what are we going to sort first? Let's start with sorting by the number of bottles sold, in ascending order. To do this, we'll need to compare the `sold` property of each object. Now, one thing you should take note of is that because this is an array of product objects, the compare function is going to be passed two objects, not two numbers:

compareSold takes two cola product objects and compares the sold property of colaA to the sold property of colaB.

```
function compareSold(colaA, colaB) {
    if (colaA.sold > colaB.sold) {
        return 1;
    } else if (colaA.sold === colaB.sold) {
        return 0;
    } else {
        return -1;
    }
}
```

This function will make the sort method sort the colas by number of bottles sold in ascending order.

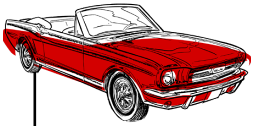
Feel free to simplify this code like you did in the earlier exercise if you want!

And of course, to use the `compareSold` function to sort the `products` array, we simply call the `products` array's `sort` method:

```
products.sort(compareSold);
```

Remember that the sort method can be used for any kind of array (numbers, strings, objects), and for any kind of sort (ascending or descending). By passing in a compare function, we get flexibility and code reuse!

Note: we're sorting ascending this time, rather than descending, so notice the difference with our previous compare function.



Test Drive

Time to test the first Webville Cola code. You'll find all the code from the last few pages consolidated below, along with some extras to test this out properly. So, just create a simple HTML page with this code ("cola.html") and do some QA:

```
let products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];

function compareSold(colaA, colaB) {
    if (colaA.sold > colaB.sold) {
        return 1;
    } else if (colaA.sold === colaB.sold) {
        return 0;
    } else {
        return -1;
    }
}
```

Here's the compare function we'll pass to sort...

...and here's a new function we wrote to print the products so they look nice in the console. (If you just write console.log(products), you can see the output, but it doesn't look very good.)

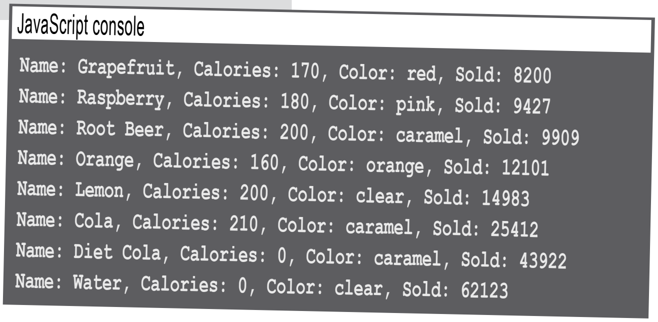
```
function printProducts(products) {
    for (let i = 0; i < products.length; i++) {
        console.log(`Name: ${products[i].name},
                     Calories: ${products[i].calories},
                     Color: ${products[i].color},
                     Sold: ${products[i].sold}`);
    }
}
```

So, we first sort the products, using compareSold...

```
products.sort(compareSold);
printProducts(products);
```

...and then print the results.

Here's our output running the code using the compareSold sort function. Notice the products are in order by the number of bottles sold.



JavaScript console

```
Name: Grapefruit, Calories: 170, Color: red, Sold: 8200
Name: Raspberry, Calories: 180, Color: pink, Sold: 9427
Name: Root Beer, Calories: 200, Color: caramel, Sold: 9909
Name: Orange, Calories: 160, Color: orange, Sold: 12101
Name: Lemon, Calories: 200, Color: clear, Sold: 14983
Name: Cola, Calories: 210, Color: caramel, Sold: 25412
Name: Diet Cola, Calories: 0, Color: caramel, Sold: 43922
Name: Water, Calories: 0, Color: clear, Sold: 62123
```

Exercise

Now that we have a way to sort colas by the sold property, it's time to write compare functions for each of the other properties in the product object: name, calories, and color. Check the output you see in the console carefully; make sure for each kind of sort, the products are sorted correctly. Check your answer at the end of the chapter.

Write your solutions for the remaining three sort functions below.

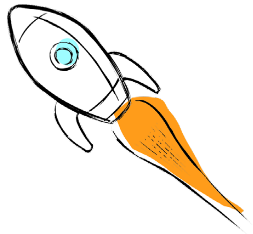
Hint: you can use <, >, and === to sort alphabetically too!

```
function compareName(colaA, colaB) {




}
function compareCalories(colaA, colaB) {




}

function compareColor(colaA, colaB) {




}
products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

For each new compare function, we call sort and display the results in the console.

Solution on page 487

Introducing higher-order functions



We already know that JavaScript functions are first class values; we can pass then around and do anything with them we'd do with any other JavaScript type, whether that is storing them in a variable, passing them to a function, or returning them from a function. In fact, we have a name for functions that take one or more functions as arguments, or return a function (or both): *higher-order functions*.

You might be asking, what is the point? Well, we'll return to the topic of returning functions from functions, but when we're talking about passing functions to functions, why would we want to do that? Isn't the point of a function to compute something?

Think about it like this: we pass parameters to functions to *parameterize* them. We're used to thinking about parameterizing functions with different values. If we have a function that draws buttons, we might parameterize the size, the label of the button, and the color, so one function can create many kinds of buttons.

So what are we parameterizing by passing a function to another function? How about *computation*? For instance, as we just saw, say we have a function that sorts arrays, and we'd like to tell it *how* it should compare the items in the array it is sorting. We can do that by passing a function to the array's `sort` method that computes a comparison of an array's items. That's a higher-order function!

There are a few other higher-order functions you're going to see a lot in JavaScript. These too work well with arrays. Let's check them out:

☐ **map**: `map` does some computation on each item in the array, mapping each value to a return value. The result is a new array of those return values.

☐ **filter**: `filter` tests each item with some computation (which returns true or false). If the result is true (the item passes the test), the item is filtered into a new array.

☐ **reduce**: `reduce` reduces an array to one value, using some computation and an accumulator.

☐ **forEach**: `forEach` is another way to iterate over an array and do some computation on each item in the array.

Higher-order functions allow you to parameterize computation, which can provide a high degree of flexibility and reusability in your code.

Filtering with higher-order functions

Let's put our knowledge to use and filter Webville Cola's products with the array's `filter` method. To use `filter`, we write a function that performs a test on one array item and determines if it passes that test. We pass that function to the array's `filter` method, which applies the function to every item in the array. If an item passes the test, that item is included in a new array created by `filter`; otherwise, it is excluded.

Let's say Webville Cola is interested to know how many of its products are actually colas, as opposed to fruit-flavored drinks, water, or root beer. First, we'll write a function to see if the name of a given cola from the `products` array includes the string "Cola" (this is our test):

```
let isCola = function(cola) {
    return (cola.name.includes("Cola"));
};
```

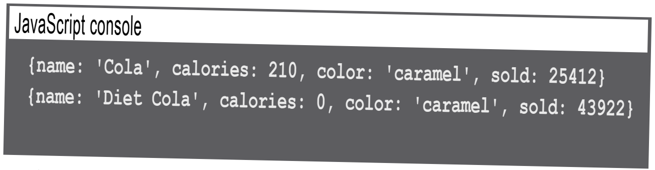
Our function takes a product and tests to see if the name includes "Cola".

We can now use our `isCola` function to filter out all the non-cola products in the `products` array and give us a new array containing only the colas:

```
let colas = products.filter(isCola);
console.log(colas);
```

We pass the isCola function to the filter method, which applies it to every drink in the products array.

Remember that unlike with the sort method, which destructively alters its array, with filter we're getting a brand-new array back; the original array remains unchanged.



```
{name: 'Cola', calories: 210, color: 'caramel', sold: 25412}
{name: 'Diet Cola', calories: 0, color: 'caramel', sold: 43922}
```

Two colas matched the test and are returned in a brand-new array.

Exercise

It's your turn. Write a test function to use with filter that filters products that have sold more than 20,000 bottles. Check our solution at the end of the chapter before you go on.

Solution on page 488

Don't forget your anonymous and arrow functions

So far, our code looks like this:

```
let isCola = function(cola) {
    return (cola.name.includes("Cola"));
};
let colas = products.filter(isCola);
```

We define a function here...

...and pass it to the filter method here.

You'll often see anonymous and arrow functions used when you need just a tiny bit of computation in a function. We call these "right here, right now functions," and you'll often see them used anywhere you encounter a method that expects a function and that function is just one or two lines of code.

Now, ask yourself this: do we need the `isCola` function anywhere but with this one filter? Probably not. Maybe we don't need to define the function separately, and we can use an anonymous function instead:

```
colas = products.filter(function(cola) {
    return (cola.name.includes("Cola"));
});
```

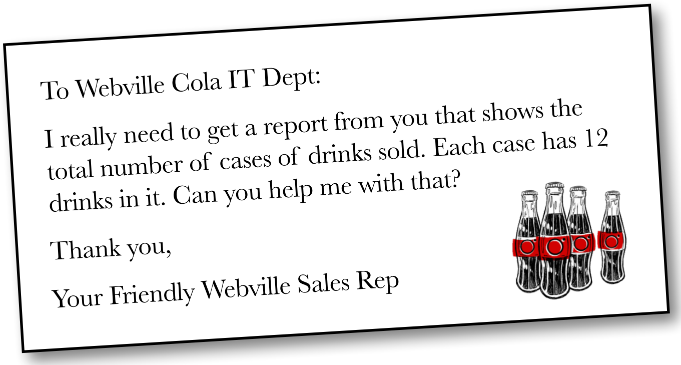
Here, we put the anonymous function definition right in the call to filter.

Hmmm...can we simplify this even further, with an arrow function?

```
colas = products.filter((cola) => cola.name.includes("Cola"));
```

And here we make the function even more concise by using an arrow function. Exactly what arrow functions are designed for.

Wow, we got our code down to one line!



To Webville Cola IT Dept:

I really need to get a report from you that shows the total number of cases of drinks sold. Each case has 12 drinks in it. Can you help me with that?

Thank you,

Your Friendly Webville Sales Rep

Just in...

It looks like we have a new request from the Webville Cola sales representative to create a report that shows the total number of *cases* of drinks sold. Currently, each product object has the total number of *individual* drinks sold.

Can we use higher-order functions to help? Yes! Let's see how.

Serious Coding

One more arrow function rule: if you have only one argument, you can skip the parentheses on the left side of the arrow. Like this:

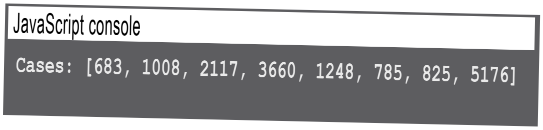
```
let addOne = n => n + 1;
```

How to get cases sold with map

The first thing we can do is convert the number of drinks sold for each product to the number of cases sold, by dividing each `sold` value by 12. We could write a for loop to do this... but instead, let's use the higher-order array method `map`, like this:

```
let cases = products.map(cola => Math.floor(cola.sold/12));
console.log("Cases:", cases);
```

Note we're using Math.floor to make sure we're getting whole numbers of cases.



```
Cases: [683, 1008, 2117, 3660, 1248, 785, 825, 5176]
```

Here's the array of cases sold for all the products in the products array. The first item is the cases of grapefruit, the second is the cases of orange, and so on.

The `map` method takes a function that's applied to each drink in the `products` array and returns a value that is that drink's `sold` value divided by 12. `map` takes all the values returned and gathers them up into a new array of the number of cases sold for each product in the `products` array.

In other words, we're mapping an array of products to an array of cases sold.

Now that we have the cases sold for all our products, how can we turn that into the total number of cases sold?

Get the total cases sold with reduce

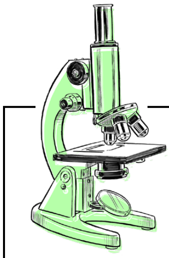
Again, we could write a for loop to iterate through the `cases` array and add up the number of cases sold, but let's use the higher-order array method, `reduce`, instead.

Like `filter` and `map`, `reduce` takes a function that is applied to each item in an array. However, it's slightly different from `filter` and `map` in that it also takes a running total that gets accumulated each time the function is applied to another item in the array. That's because the whole point of `reduce` is to turn the array into one value. In our case, that one value should be the total number of cases.

The array of cases of drinks sold.

```
[683, 1008, 2117, 3660, 1248, 785, 825, 5176]
```

What we want is to add all these up and produce a total number of cases sold.



Anatomy of the reduce Method

Let's take a closer look at the `reduce` method to see how it works before we use it in our Webville Cola code:

```
let myArray = [1, 2, 3];
```

We have an array, and we want to add up the numbers in the array to get a total.

reduce is a method of an array.

It takes two arguments: a reducer function that is applied to each element in the array, and an initial value for the accumulator.

```
let total = myArray.reduce((acc, v) => acc + v, 0);
```

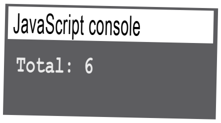
The reducer function takes two values: an accumulator and a value from the array. The accumulator is the return value from the computation on the previous array item.

We want the total of adding up all the values in the array, so each time the reducer function is called we add the value to the accumulator.

There is no "return value from the computation on the previous array item" for the first item in the array, so we supply an initial value for the accumulator as the second argument to reduce.

The total of adding up the values in the array is returned once reduce is completed, and we store that value in total and display it in the console.

```
console.log("Total:", total);
```



JavaScript console

```
Total: 6
```

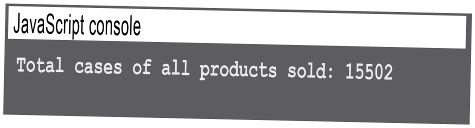
Using reduce to get the total cases sold

Okay, now that we know how `reduce` works, let's get that total number of cases sold for the Webville Cola sales rep. We'll reduce the array of cases we got back using the `map` method, adding each new value of cases sold to the total sold accumulator, and store the result in `totalCases`, like this:

For each item in the cases array, we add the number of cases to the accumulator, which we start out at 0.

```
let totalCases = cases.reduce((totalSold, cases) => totalSold + cases, 0);
console.log("Total cases of all products sold:", totalCases);
```

And...the final total is...15,502 cases of Webville Cola products sold! The sales rep will be happy.



```
Total cases of all products sold: 15502
```

Exercise

It's your turn. Write code using array methods to compute the total number of calories of "caramel" colored products. Check our solution at the end of the chapter before you go on.

Your code here.

Solution on page 488

there are no Dumb Questions

Q: Can I use reduce with things other than numbers, like, say, strings?

A: Yes, you can. You can use reduce with an array of any type of values to reduce that array to one thing. You might have to think carefully about what you use as the starting value for the accumulator, though, depending on what you want your reducer function to do.

Exercise

This doesn't really seem like anything Webville Cola might want to do, but for fun, can you write code using the reduce array method to create a string out of the first character of each of the product names? That is, your code should produce the string "GOCDLRRW". Check our solution at the end of the chapter before you go on.

Your code here.

Solution on page 488

I can think of a lot of cases where we might use all of these, like mapping something and then filtering it and then reducing it, and so on.

You're right, this is a very common pattern. In fact, it is so common that the design of JavaScript's higher-order functions allows for this. What we can do is link together some of these requests into a sequence of function calls that all work together. It's a technique called *chaining*, and it looks like this:

```
products.map(fun1).filter(fun2).reduce(fun3);
```

array array array value →

The `map`, `filter`, and `reduce` functions all allow chaining. Let's get some experience doing this with Webville Cola.

Chaining map, filter, and reduce

With chaining, we take the result of one operation and call another operation on that result, all in one long statement. Let's see how we can chain together the `map` and `reduce` operations we just did on the Webville Cola products array to streamline our code.

Right now, we've got two statements to find the total cases sold; the first maps the products array to a cases array, and the second reduces the cases array to the total cases sold:

Notice, we're getting back an array of cases here...

```
let cases = products.map(cola => Math.floor(cola.sold/12));
let totalCases = cases.reduce((totalSold, cases) => totalSold + cases, 0);
```

...and immediately using that array in the second statement for reduce.

We can combine these two statements by chaining them together.

Chained together, these two statements now look like this:

```
let totalCasesSold =
    products.map(cola => Math.floor(cola.sold/12))
            .reduce((totalSold, cases) => totalSold + cases, 0);
```

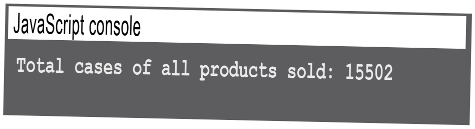
First we do the map...

...and then the reduce.

We're using two lines to write this, but note we're calling reduce on the array resulting from map.

```
console.log("Total cases of all products sold:", totalCasesSold);
```

And the result is the same.



JavaScript console

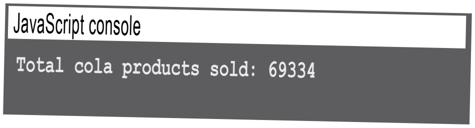
```
Total cases of all products sold: 15502
```

What about filter; can we chain that too? Yes. Check out how we do that in this code to compute the total number of colas sold:

First we filter only the drinks with "Cola" in the name into an array...

```
let totalColasSold =
    products.filter(cola => cola.name.includes("Cola"))
            .reduce((totalSold, cola) => totalSold + cola.sold, 0);
console.log("Total cola products sold:", totalColasSold);
```

...and then we reduce that array to the total colas sold.



JavaScript console

```
Total cola products sold: 69334
```

Exercise

It's your turn. Write a chain combining array methods to compute the total number of drinks sold that have less than 200 calories each. Check our solution at the end of the chapter before you go on.

Solution on page 488

Who Does What?

Match the terms on the left to the descriptions on the right! Check your answers before moving on.

map

filter

reduce

forEach

Does some computation on each item in the array and creates a new array with the resulting values.

Tests each item in an array with a boolean test; if the test is true, the item is added to a new array that is returned.

Collapses an array to one value, using some computation and an accumulator.

Iterates over an array and does some computation on each item in the array.

Solution on page 488

forEach? We never really talked about forEach.

You got us. The `forEach` array method is one more higher-order function you should know about. It gives us a way to iterate over an array and supply a function that does something to each item in the array. So unlike with, say, a for loop, we don't specify any actual iteration; we just supply the function and `forEach` takes care of the rest. Let's give it a try.

Iterating with forEach

Remember our `printProducts` function from earlier in the chapter? We wrote this function to print out all the Webville Cola products in the products array:

```
function printProducts(products) {
    for (let i = 0; i < products.length; i++) {
        console.log(`Name: ${products[i].name},
                     Calories: ${products[i].calories},
                     Color: ${products[i].color},
                     Sold: ${products[i].sold}`);
    }
}
printProducts(products);
```

Here, we used a standard for loop to iterate over the products array and display each item in the console.

Note we're displaying each value on separate lines here, but they can be joined together in one line instead if you prefer.

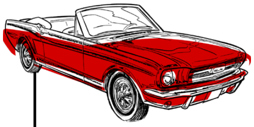
We call the function and pass in the products array.

With the `forEach` method, we can do the same thing in one statement, like this:

```
let products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];


products.forEach(cola =>
    console.log(`Name: ${cola.name},
                 Calories: ${cola.calories},
                 Color: ${cola.color},
                 Sold: ${cola.sold}`)
);
```

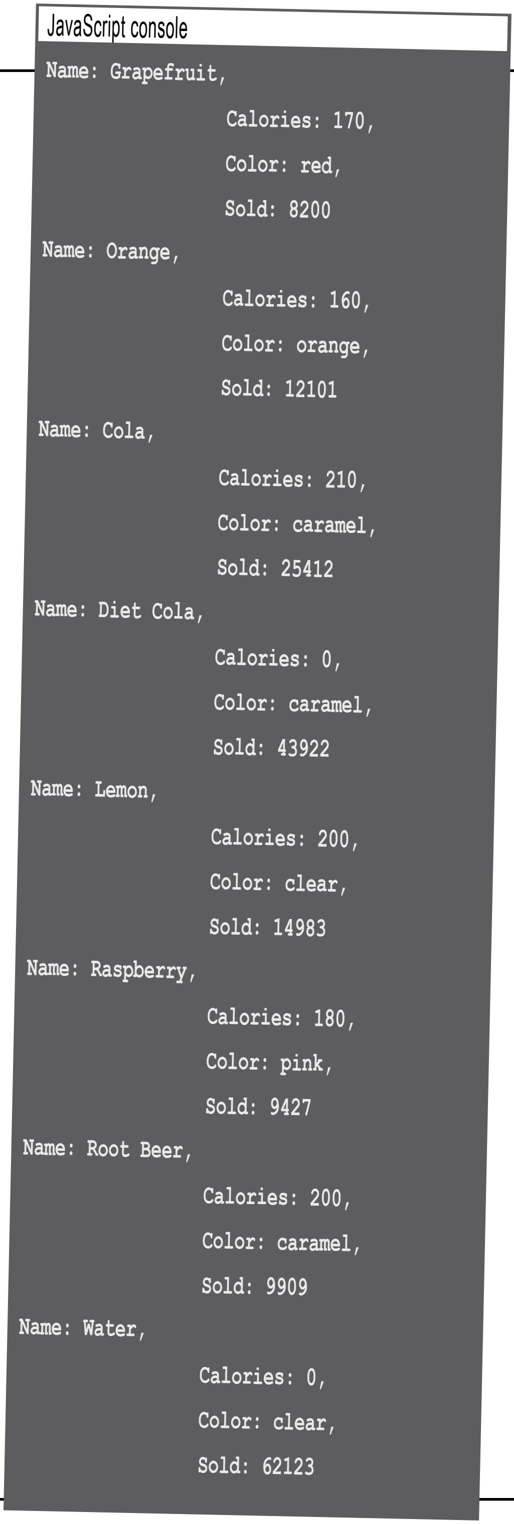
We're passing a function to the forEach method that is applied to each item in the products array. We just display the item in the console, like before (one value on each line, but you can also combine them on one line if you like).

Try it!



Test Drive

Say bye-bye to all that for (let i = 0; i < arr.length; i++) syntax. With forEach we can write a nice, clean iteration just by passing a function.



JavaScript console

```
Name: Grapefruit,
                    Calories: 170,
                    Color: red,
                    Sold: 8200
Name: Orange,
                    Calories: 160,
                    Color: orange,
                    Sold: 12101
Name: Cola,
                    Calories: 210,
                    Color: caramel,
                    Sold: 25412
Name: Diet Cola,
                    Calories: 0,
                    Color: caramel,
                    Sold: 43922
Name: Lemon,
                    Calories: 200,
                    Color: clear,
                    Sold: 14983
Name: Raspberry,
                    Calories: 180,
                    Color: pink,
                    Sold: 9427
Name: Root Beer,
                    Calories: 200,
                    Color: caramel,
                    Sold: 9909
Name: Water,
                    Calories: 0,
                    Color: clear,
                    Sold: 62123
```

I have to admit this higher-order function stuff is pretty powerful.

Bullet Points

- There are three ways to define a function: with a **function declaration**, with a **function expression**, or with an **arrow function**.
- A **function reference** is a value that refers to a function.
- When the browser evaluates a function declaration, it creates a function as well as a variable with the same name as the function, and stores the function reference in the variable.
- When the browser evaluates a function expression or arrow function, it creates a function, and it's up to you what to do with the function reference.
- A function without a name is known as an **anonymous function**.
- We often use anonymous functions as event handlers or to pass as one-off functions to other functions, like setTimeout.
- **First class** values can be assigned to variables, included in data structures, passed to functions, or returned from functions.
- Functions are first class values.
- The goal of arrow functions is conciseness and expressivity.
- **Arrow functions** are very similar to function expressions, except they generally shouldn't be used as object methods or event handlers because the value of "this" will not be set to what you expect (i.e., the object whose method you're calling or whose event you're handling).
- A **higher-order function** is a function that takes a function and/or returns a function.
- The array method **sort** is a higher-order function that takes a function that knows how to compare two values in an array.
- The function you pass to the sort method should return one of these values: a number greater than 0, 0, or a number less than 0. The sort method uses these values to sort an array in ascending or descending order.
- Other higher-order functions that are array methods include map, filter, reduce, and forEach.
- The **map** method takes a function that is applied to every item in the array and returns a new value that's added to a new array. In other words, it maps from one array of values to another.
- The **filter** method takes a function that is applied to every item in the array and returns true or false. If true is returned, then the value from the original array is added to a new array. In other words, it filters all items from an array that pass a test into a new array.
- The **reduce** method takes a function that is applied to every item in the array and returns one value that is the result of accumulating results from the computation on each item in the original array. In other words, it reduces an array of values to one value.
- The **forEach** method takes a function that is applied to every item in the array. It is a way of iterating over an array and performing computation on each item in the array.
- The map, filter, and reduce methods are designed so they can be **chained** together into one long statement if so desired (without the need for intermediate variables).

Sharpen your pencil Solution

From page 451

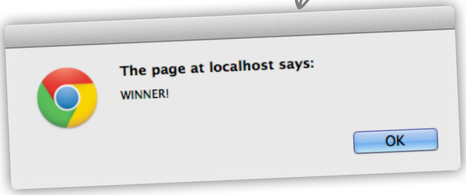
To get the idea of functions as values into your brain, let's play a little game of chance. Try the shell game. Did you win or lose? Give it a try and find out. Here's our solution.

```
let winner = function() { alert("WINNER!") };
let loser = function() { alert("LOSER!") };
// let's test as a warm up
winner();
// let's assign to other variables for practice
let a = winner;
let b = loser;
let c = loser;
a();
b();
// now let's try your luck with a shell game
c = a;
a = b;
b = c;
c = a;
a = c;
a = b;
b = c;
a();
```

Remember, these variables hold references to the winner and loser functions. We can assign and reassign these references to other variables, just like with any value.

And at any time, we can invoke a reference to a function.

c is winner
a is loser
b is winner
c is loser
a is loser
a is winner
b is loser
invoking a...
winner!!!


The page at localhost says:

WINNER!

Sharpen your pencil Solution

From page 454

Use your knowledge of functions and variables and check off the true statements below. Here's our solution.

- ☑ The handler variable holds a function reference.
- ☑ When we assign handler to window.onload, we're assigning it a function reference.
- ☑ The only reason the handler variable exists is to assign it to window.onload.
- ☑ We'll never use handler again, as it's code that is meant to run only when the page first loads.
- ☑ Invoking onload handlers twice is not a great idea—doing so could cause issues given that these handlers usually do some initialization for the entire page.
- ☑ Function expressions create function references.
- ☑ Did we mention that when we assign handler to window.onload, we're assigning it a function reference?

Sharpen your pencil Solution

From page 460

Rewrite the cookie alarm function as an arrow function in the space below. Here's our solution.

Current version of the function.

```
setTimeout(function() { alert("Time to take the cookies out of the oven");}, 600000);
```

Write the equivalent arrow function here.

```
setTimeout( () => alert("Time to take the cookies out of the oven") , 600000);
```

Sharpen your pencil Solution

From page 467

Here's our solution for comparing two numbers using much less code.

```
function compare(num1, num2) {
    return num2 - num1;
}
```

We can make this function a single line of code by simply returning the result of subtracting num1 from num2. Run through a couple of examples to see how this works. And remember, sort is expecting a number greater than, equal to, or less than 0, not specifically 1, 0, or –1 (although you'll see a lot of code return those values for sort).

From page 456

Sharpen your pencil Solution

There are a few opportunities in the code below to take advantage of anonymous functions. Go ahead and rework the code to use anonymous functions wherever possible. You can scratch out the old code and write in new code where needed. Oh, and one more task: circle any anonymous functions that are already being used in the code. Here's our solution.

```
window.onload = function() {
    let button = document.getElementById("bake");
    button.onclick = function() {
        console.log("Time to bake the cookies.");
        cookies.bake(2500);
    };
};
```

Here, we got rid of the init function declaration and turned it into an anonymous function assigned directly to the onload property of window.

We also turned the handleButton function into an anonymous function and assigned it directly to the onclick property of the button element.

```
let cookies = {
    instructions: "Preheat oven to 350...",
    bake(time) {
        console.log("Baking the cookies.");
        setTimeout(done, time);
    }
};
```

No changes here.

```
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    let cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
};
```

Here, we already have an anonymous function expression assigned to the cool variable...

...which we then pass to setTimeout here.

Exercise Solution

From page 470

Now that we have a way to sort colas by the sold property, it's time to write compare functions for each of the other properties in the product object: name, calories, and color. Check the output you see in the console carefully; make sure for each kind of sort, the products are sorted correctly. Here's our solution.

Here's our implementation of each compare function.

```
function compareName(colaA, colaB) {
    if (colaA.name > colaB.name) {
        return 1;
    } else if (colaA.name === colaB.name) {
        return 0;
    } else {
        return -1;
    }
}

function compareCalories(colaA, colaB) {
    if (colaA.calories > colaB.calories) {
        return 1;
    } else if (colaA.calories === colaB.calories) {
        return 0;
    } else {
        return -1;
    }
}

function compareColor(colaA, colaB) {
    if (colaA.color > colaB.color) {
        return 1;
    } else if (colaA.color === colaB.color) {
        return 0;
    } else {
        return -1;
    }
}

products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

For each new compare function, we call sort and display the results in the console.

Exercise Solution

From page 473

It's your turn. Write a test function to use with filter that filters products that have sold more than 20,000 bottles. Here's our solution.

```
let highSales = function(cola) { return (cola.sold > 20000); };
let highSellingColas = products.filter(highSales);
console.log("High selling colas:", highSellingColas);
```

Exercise Solution

From page 476

It's your turn. Write code using array methods to compute the total number of calories of "caramel" colored products. Here's our solution.

```
let caramel = products.filter(cola => cola.color.includes("caramel"));
let cc = caramel.reduce((cals, cola) => cals + cola.calories, 0);
console.log("Caramel calories:", cc);
```

Exercise Solution

From page 476

This doesn't really seem like anything Webville Cola might want to do, but for fun, can you write code using the reduce array method to create a string out of the first character of each of the product names? That is, your code should produce the string "GOCDLRRW". Here's our solution.

```
let firstChars = products.reduce((str, cola) => `${str}${cola.name.charAt(0)}`,"");
console.log("First characters of names:", firstChars);
```

Exercise Solution

From page 479

It's your turn. Write a chain combining array methods to compute the total number of drinks sold that have less than 200 calories each. Here's our solution.

```
let totalLowerCaloriesDrinksSold =
    products.filter(cola => cola.calories < 200)
            .reduce((totalSold, cola) => totalSold + cola.sold, 0);
console.log("Total lower calorie drinks sold:", totalLowerCaloriesDrinksSold);
```

Who Does What? Solution

From page 479

Match the terms on the left to the descriptions on the right! Here's our solution.

map	Does some computation on each item in the array and creates a new array with the resulting values.
filter	Tests each item in an array with a boolean test; if the test is true, the item is added to a new array that is returned.
reduce	Collapses an array to one value, using some computation and an accumulator.
forEach	Iterates over an array and does some computation on each item in the array.

We never said they were mixed up!
But we made you think, didn't we?

11 modern syntax, lexical scope, and closures

Serious functions

DINING
NOOK
FOYER
GREAT RM.
GARAGE
MASTER

You've put functions In this chapter, we take it further; we get hardcore. We'll show you how to up your syntax game with advanced techniques for handling arguments, parameters, and assignments. We'll then take another look at scope and some of the finer points of how JavaScript manages it. This journey through the subtleties of scope brings us to the heart of closures—a concept often shrouded in mystery but pivotal in mastering JavaScript. In the end, you'll be more expressive with your JavaScript than you thought possible. through their paces, but there's more to learn.

And if you have heard of a closure but don't quite know what it is, you're in the right place!

Getting serious about functional syntax

You might think at this point there is nothing else to know about basic function syntax, but there are actually a number of syntactic enhancements that have been added to JavaScript that can make your use of functions even more straightforward and expressive. Before we get down to talking about scope and closures, let's make sure you're up to speed on everything you can do when calling and receiving values from functions.

Streamlining code with default parameters

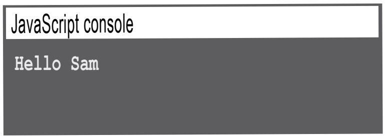
Let's say we have a function, `greeting`, that takes a message and a recipient, like this:

```
function greeting(message, recipient) {
   console.log(message, recipient);
}
```

Here we get "Hello Sam", just as you probably would have expected.

Let's call `greeting` like this:

```
greeting("Hello", "Sam");
```

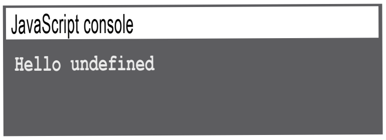


JavaScript console

```
Hello Sam
```

What do you think we get if we leave off an argument, like this?

```
greeting("Hello");
```



JavaScript console

```
Hello undefined
```

Here, things go wrong. We didn't pass a value for recipient, so the value of recipient is undefined.

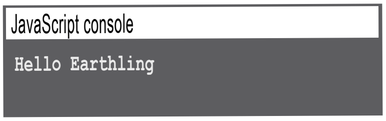
JavaScript gives us a way to deal with this by providing a default parameter value, so if the caller doesn't pass along an argument, that default value is used instead of `undefined`. It works like this:

```
function greeting(message, recipient="Earthling") {
   console.log(message, recipient);
}
```

If no argument is passed for the recipient parameter, then the string "Earthling" is used by default.

Let’s give it a try:

```
greeting("Hello");
```



```
Hello Earthling
```

Now “Earthling” is substituted for the missing argument when greeting is called.

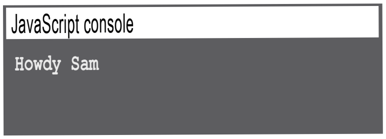
You can specify a default value for any, all, or none of the parameters. Here, we provide default values for both parameters in `greeting`:

```
function greeting(message="Howdy", recipient="Earthling") {
   console.log(message, recipient);
}
```

However, note that if you want to use the default values for parameters that are before the parameter you specify, you’ll have to pass an argument of `undefined` for those parameters:

```
greeting(undefined, "Sam");
```

We need to specify undefined for any parameter we’d like a default value for, unless it is the last parameter.



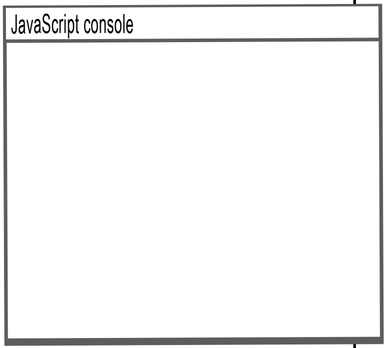
```
Howdy Sam
```

Now we are using the default value for the message parameter and the argument “Sam” for the recipient parameter.

BE the Browser

Your job is to act like you’re the browser. You need to evaluate the code and produce its output. We’ve already started it for you.

```
function pitch(noun="AI", verb="powered", obj="scooter") {
    console.log(noun, verb, obj);
}
pitch("Human", "designed", "bicycle");
pitch("Monkey", undefined, "typewriter");
pitch();
pitch("Peddle");
pitch(undefined, "designed");
pitch(undefined, undefined, "robot");
```



Your output here.

Solution on page 535

Is there a way to deal with an arbitrary number of arguments being passed to a function? I've seen that in a couple of other languages.

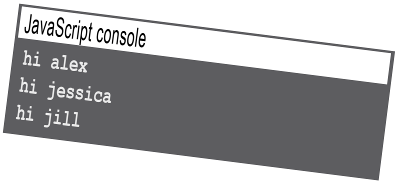
Yes. We can use rest parameters. JavaScript provides rest parameters as a way to specify an indefinite number of arguments as an array. This can be very helpful when you don't know the number of arguments a caller is going to pass to a function. Rest parameters are denoted by three dots `...` followed by the name of the array you'd like to hold the arguments, like this:

```
function greeting(message, ...recipients) {
    recipients.forEach(r => console.log(message, r));
}
```

You can choose whatever variable name you like for the rest parameter. Note that the rest parameter must be the last parameter.

Let's give the function above a test.

```
greeting("hi", "alex", "jessica", "jill");
```


JavaScript console

```
hi alex
hi jessica
hi jill
```

Sharpen your pencil

Write a function named echo that takes any number of strings and prints out each one to the console log. Check our solution at the end of the chapter.

Solution on page 535

Spreading out your arguments

Okay, we've looked at the parameter side of things; now let's take a look at a way to pass arguments to our functions using the spread operator, otherwise known as `...` (three dots). Here's how it works.

Yup, spread and rest are both operators, but we say rest when it's used with a function parameter and spread when it's used with an argument.

Say we have an array along with a function called `sum`:

```
let numbers = [1, 2, 3, 4];

function sum(a, b, c, d) {
   console.log("The answer is", a + b + c + d);
}
```

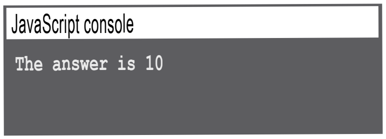
We'd like to pass each value in the `numbers` array to a different parameter in the function `sum`. You might think we have to do this:

```
sum(numbers[0], numbers[1], numbers[2], numbers[3]);
```

But with the spread operator we can expand the array in place, like this:

```
sum(...numbers);
```

This is expanded into sum(1, 2, 3, 4);.

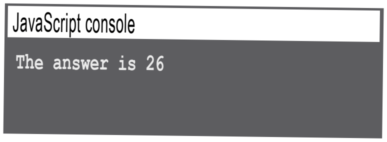

JavaScript console

```
The answer is 10
```

Unlike with rest parameters, we can use this as much as we want in a function call:

```
let values1 = [5, 6];
let values2 = [7, 8];

sum(...values1, ...values2);
```

This is expanded into sum(5, 6, 7, 8);.

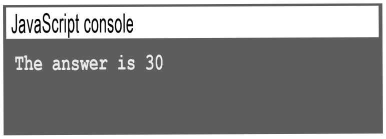

JavaScript console

```
The answer is 26
```

We can even mix spread arguments with regular arguments:

```
sum(...values1, 9, 10);
```

This is expanded into sum(5, 6, 9, 10);.


JavaScript console

```
The answer is 30
```

there are no Dumb Questions

Q: Is the spread operator just for arrays?

A: No, you can use spread with arrays, strings, the properties in an object, or any collection of values that is iterable (like Set or Map; see the appendix). In the case of strings, the spread operator breaks a character string into its individual characters.

Q: I've believe I've seen the spread operator used outside of function calls; is that possible?

A: Yes, you can use it when creating arrays as well. Say you want to use an existing array as the basis for a new one. You can use spread like this:

```
let iceCream = ["vanilla", "mint"];
let moreIceCream = [...iceCream, "peach"];
```

moreIceCream contains ["vanilla", "mint", "peach"].

Q: You said that spread works with objects. How does that work?

A: Very similar to what we just did with arrays. We can create a new object based on an existing one:

```
let obj1 = { a: 1, b: 2 };
let obj2 = { ...obj1, c: 3 };
```

obj2 is now {a: 1, b: 2, c: 3}.

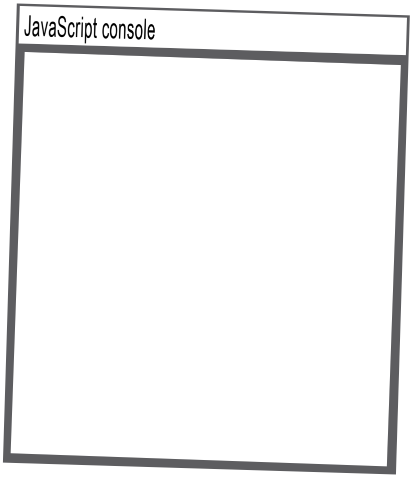
Q: Why should I care about all these fancy ways of passing arguments and specifying parameters?

A: On the surface they may seem “fancy,” but they help us write code that is more readable and that causes less bugs. Also, you’re going to be looking at a lot of code written by other software developers, and you are sure to see these techniques being used, so you want to be able to easily understand when reading someone else’s code.

Sharpen your pencil

It’s your turn. Test your function syntax chops and figure out what this code prints to the console. Write your answer in the space below, and check it at the end of the chapter before you go on.

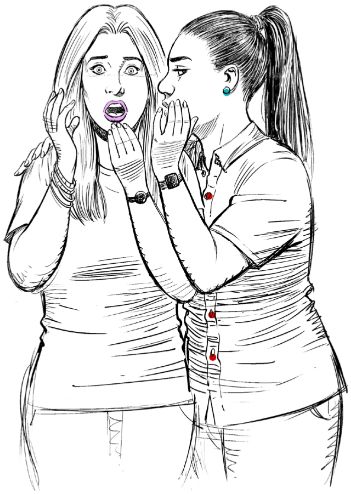
```
let arr = [1, 2, 3, 4, 5];

function foo(a = 3, b = 2, c = 1) {
    output = `${a} ${b} ${c}`;
    console.log(output);
}

function bar(a, b, ...c) {
    foo(...c);
}

bar(...arr);
bar(1, 2);
bar();
```



Write what you think this code prints to the console here.

Solution on page 536

There's something we haven't told you about functions...



In the previous chapter, we looked at several different ways to define functions:

1. We can define a function with a function declaration, like this:

```
function quack(num) {
   for (let i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
```

2. We can define a function with a function expression and assign it to a variable, like this:

```
let fly = function(num) {
   for (let i = 0; i < num; i++) {
      console.log("Flying!");
   }
};
```

3. We can define a function as an anonymous function expression, just where we need it, like this:

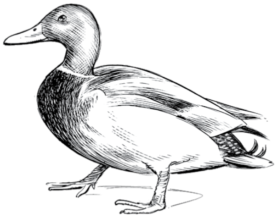
```
setTimeout(function() {
                alert("Time to take the cookies out of the oven");
            }, 600000);
```

4. And we can define a function as an arrow function, like this:

```
colas = products.filter((cola) => cola.name.includes("Cola"));
```

What we haven't told you yet is that one of these is handled a bit differently when the browser loads your page and begins to evaluate your code. One of these is a bit special, because the browser looks for it and defines these functions before it does anything else with your code. Which one? Turn the page and find out...

Function declarations are "hoisted"



Pass 1

When the browser parses the code in your page, it makes *two passes over your code.*

Here's how *hoisting* works: on the first pass—before it evaluates any code—the browser looks for function declarations. When the browser finds one, it creates a function and assigns the resulting reference to a variable with the same name as the function.

Once the browser has completed the first pass through the code and defined every function created with a function declaration, it's time for the second pass through the code. On this second pass, the browser evaluates the code, assigning values to variables, executing statements, and invoking functions.

To see how this works, let's revisit our `fly` and `quack` functions and add a bit of code to use them. We'll take two passes over the code, just like the browser does.

Pass 1:

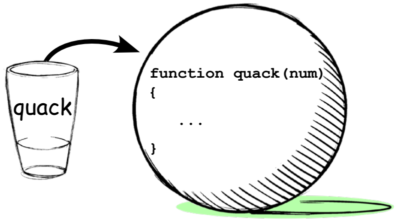
This variable declaration isn't a function declaration, so let's keep moving and ignore it for now.

```
let migrating = true;


let fly = function(num) {
   for (let i = 0; i < num; i++) {
      console.log("Flying!");
   }
};


function quack(num) {
   for (let i = 0; i < num; i++) {
      console.log("Quack!");
   }
}


if (migrating) {
     quack(2);
     fly(3);
}
```

Next, we have a function expression in a statement. That's not a declaration... let's keep moving.

Ah, here's a function declaration. We need to deal with that. Let's define the function and stash a reference to it in a variable named quack.

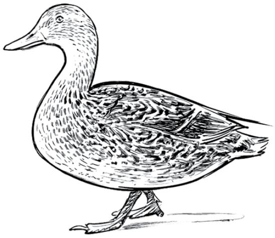
This is our hoisted function, quack.



After we deal with the declaration above, we also won't care about this code because there aren't any function declarations here.

We've done the function declarations; now we do everything else

We've completed the first pass through the code, and it's time for the second pass. Let's see what happens.



Pass 2:

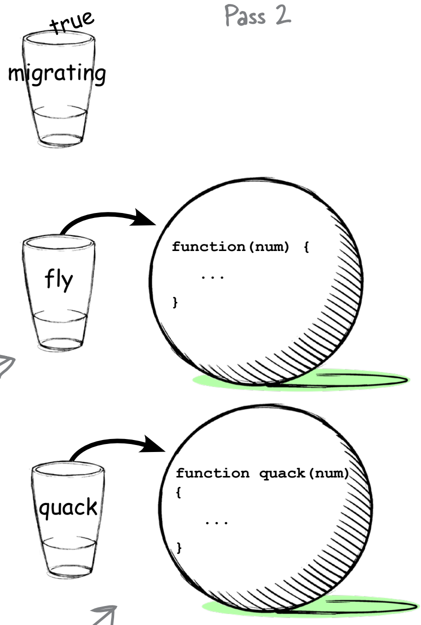
```
let migrating = true;

let fly = function(num) {
    for (let i = 0; i < num; i++) {
        console.log("Flying!");
    }
};

function quack(num) {
    for (let i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(2);
    fly(3);
}
```

Ah, time to create the variable and assign a value to it.

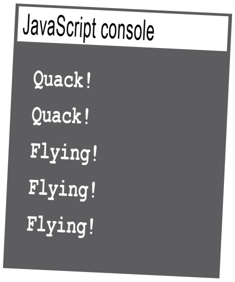
Another variable to assign. This one is called fly, and its value is a function. So, I'll create a function and assign its reference to fly.

I already did this on the first pass, so I don't have to do anything here. The quack function has been defined and its reference assigned to the variable quack.

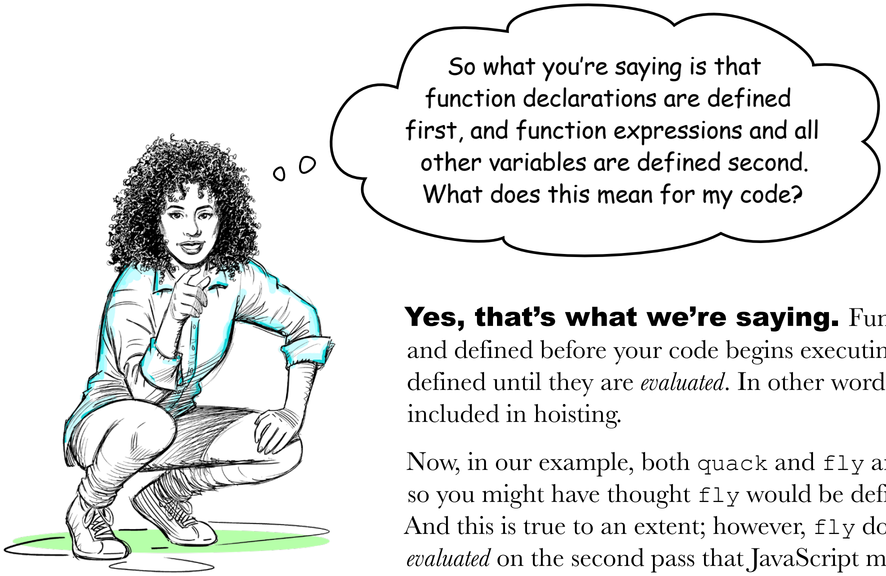
Now it's time to execute this code. migrating is true, so I'll call quack and pass the value 2 to its num parameter; and I'll call fly and pass 3 to its num parameter.



And here's the output...



```
JavaScript console
Quack!
Quack!
Flying!
Flying!
Flying!
```


So what you're saying is that function declarations are defined first, and function expressions and all other variables are defined second. What does this mean for my code?

Yes, that's what we're saying. Function declarations are hoisted and defined before your code begins executing; function expressions aren't defined until they are *evaluated*. In other words, function expressions aren't included in hoisting.

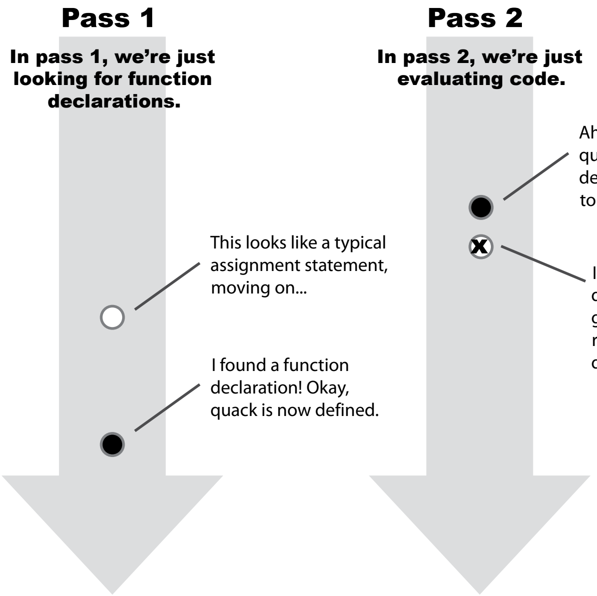
Now, in our example, both `quack` and `fly` are essentially global variables, so you might have thought `fly` would be defined everywhere, like `quack`. And this is true to an extent; however, `fly` doesn't exist until it is actually *evaluated* on the second pass that JavaScript makes through your code.

So what does this all mean? For starters, it means that you can place top-level (i.e., global) function declarations anywhere in your code—at the top, at the bottom, in the middle—and invoke them wherever you like. However, function expressions are only callable after they have been evaluated.

To see this for yourself, try moving the **if** statement *above* where you define `fly` and `quack`. Let's see what happens:

We've moved the if statement above where we define fly and quack.

```
let migrating = true;

if (migrating) {
    quack(2);
    fly(3);
}
let fly = function() {
   console.log("Flying!");
};
function quack(num) {
   console.log("Quack!");
}
```



Pass 1

In pass 1, we're just looking for function declarations.

This looks like a typical assignment statement, moving on...

I found a function declaration! Okay, quack is now defined.

Pass 2

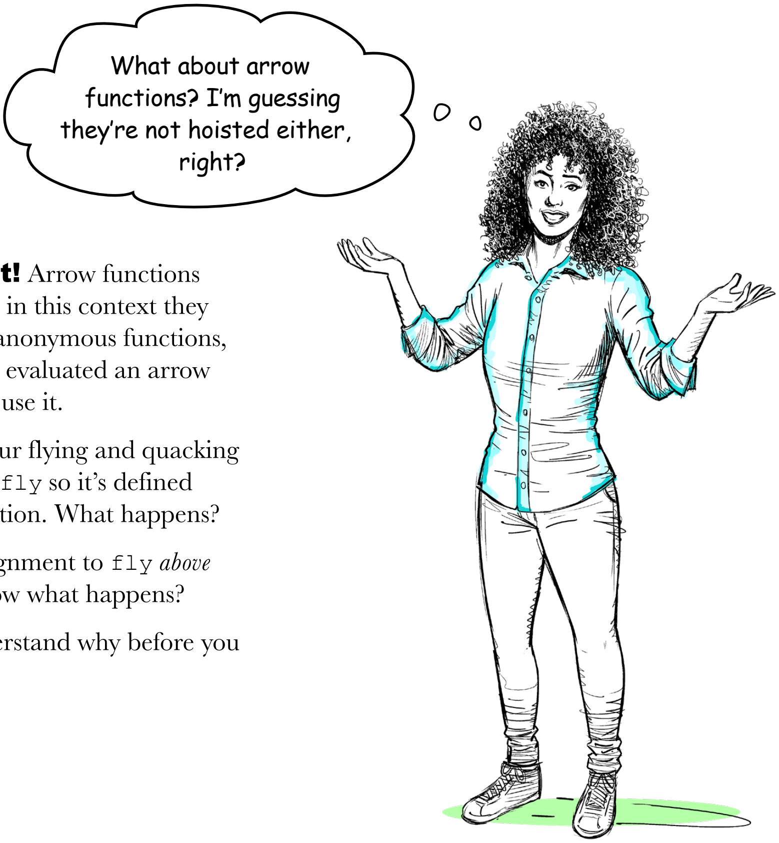
In pass 2, we're just evaluating code.

Ah, we have a call to quack; I have that all defined and ready to call.

I don't see any definition of fly; I'm going to have to report a "fly is not defined" error.

Now we'll get an error because we're trying to invoke fly before it's been defined.

Try it!


What about arrow functions? I'm guessing they're not hoisted either, right?

You are correct! Arrow functions are also not hoisted; in this context they behave exactly like anonymous functions, so make sure you've evaluated an arrow function before you use it.

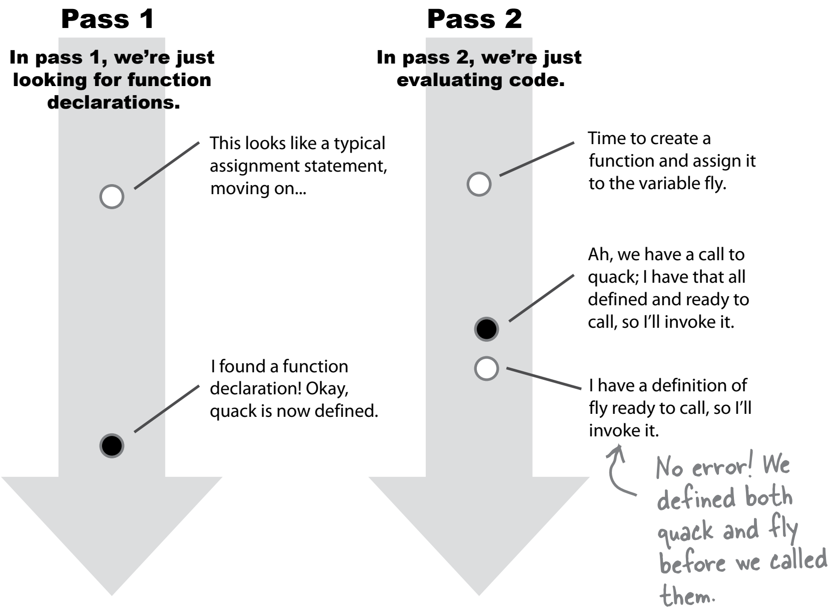
Give it a try with your flying and quacking code. Try changing `fly` so it's defined using an arrow function. What happens?

Try moving the assignment to `fly` *above* the **if** statement. Now what happens?

Make sure you understand why before you go on.

We've changed the fly variable so it's assigned an arrow function. And we've moved it above the if statement.

```
let migrating = true;

let fly = () =>
   console.log("Flying!");

if (migrating) {
     quack(2);
     fly(3);
}
function quack(num) {
   console.log("Quack!");
}
```



Pass 1

In pass 1, we're just looking for function declarations.

This looks like a typical assignment statement, moving on...

I found a function declaration! Okay, quack is now defined.

Pass 2

In pass 2, we're just evaluating code.

Time to create a function and assign it to the variable fly.

Ah, we have a call to quack; I have that all defined and ready to call, so I'll invoke it.

I have a definition of fly ready to call, so I'll invoke it.

No error! We defined both quack and fly before we called them.

BE the Browser

Below, you'll find JavaScript code. Your job is to play like you're the browser evaluating the code. In the space to the right of the code below, record each function as it gets created. Remember to make two passes over the code: the pass that processes declarations, and the second pass that handles expressions.

Write, in order, the names of the functions as they are created. If a function is created with a function expression, put the name of the variable it is assigned to. We've done the first one for you.

```
let midi = true;
let type = "piano";
let midiInterface;

function play(sequence) {
    // code here
}
let pause = function() {
    stop();
}
function stop() {
    // code here
}

function createMidi() {
   // code here
}

if (midi) {
    midiInterface = (type) => {
        // code here
    };
}
```

play

Solution on page 537

We need to talk about scope

Perhaps we've become a little too relaxed about JavaScript scope. Sure, we were on it earlier in the book, but now with all this hoisting, and with these anonymous and arrow functions added to the mix, we *really need* to understand scope at a deeper level. We blew through the last chapter assuming we could handle scope just like our standard functions, but as you'll see, there are times when we need to pay careful attention to scope.

Why all this focus on scope? Well, we've reached that point in our journey where truly understanding scope is going to have some *big payoffs*. Let's start by exploring why functions don't necessarily need to have a global scope.

Taking functions beyond global scope

Functions don't always have to live at the global level. Did you know you can actually nest functions inside other functions? Check it out:

```
function outerFunction() {
    let outerVariable = "I'm outside!";

    function innerFunction() {
        let innerVariable = "I'm inside!";
        console.log(innerVariable);
        console.log(outerVariable);
    }

    console.log(outerVariable);
    innerFunction();
}
```

Here's our first function, outerFunction.

We also have a local variable, outerVariable, defined.

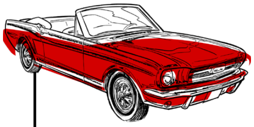
And here is another function declaration that is actually nested inside outerFunction.

Notice that innerFunction has its own local variable too.

After declaring the function, outerFunction prints out its own local variable...

...and then calls innerFunction.

Trace this code before you look at the output in the Test Drive below. What do you think this code outputs?



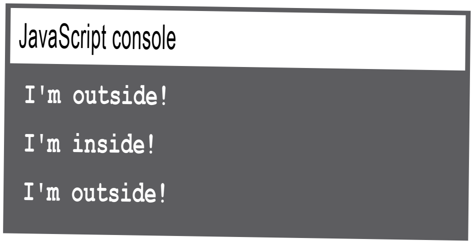
Test Drive

Here's our output. Does it match what you expected?

Call the outermost function like this.

```
outerFunction()
```

outerFunction first prints its local variable, "I'm outside", then calls innerFunction, which prints both its local variable and outerFunction's local variable.



JavaScript console

```
I'm outside!
I'm inside!
I'm outside!
```

What would happen if we tried to print innerVariable from outerFunction? Let's try it on the next page.

Exercise

Let's make a code change and have outerFunction also print out innerVariable. Carefully trace the code to see what you think the output will be. Then go ahead and make the code change and run the code.

```
function outerFunction() {
    let outerVariable = "I'm outside!";

    function innerFunction() {
        let innerVariable = "I'm inside!";
        console.log(innerVariable);
        console.log(outerVariable);
    }

    console.log(outerVariable);
    console.log(innerVariable);
    innerFunction();
}

outerFunction();
```

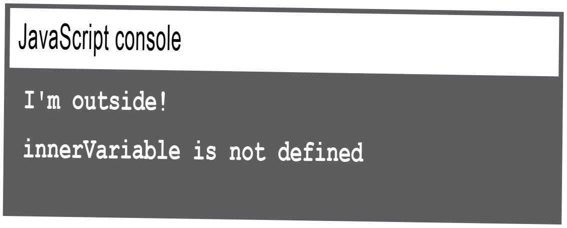
Let's try to print innerVariable from outerFunction. Do you think it will work? Give it a try.

Test Drive

Here's the output for this exercise.

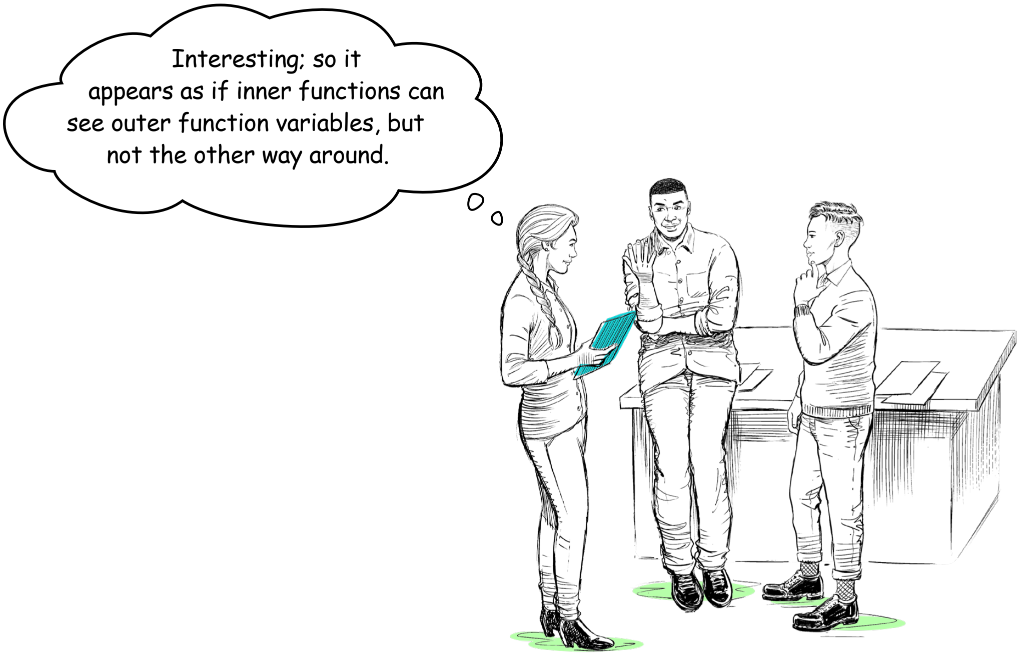
Okay, that didn't go well. When we try to print innerVariable from outerFunction, we get an error: the variable isn't defined.

So, within outerFunction, innerVariable must be out of scope. We probably need to think this through.



JavaScript console

```
I'm outside!
innerVariable is not defined
```


Interesting; so it appears as if inner functions can see outer function variables, but not the other way around.

Mara: I'm remembering from Chapter 3 that variables inside a block are visible only in that block and not outside the block.

Sam: But this is a function.

Mara: Right, but all functions have a code block.

Sam: Ah, right.

Mara: So in this case, in the inner function, we can see any local variables within that function, or any variables in the outer function. But from the outer function, we can only see the outer function's local variables, not down into the inner function.

Sam: Based on what we said before, that sounds right to me.

Alex: Isn't this called lexical scope?

Sam: What's that?

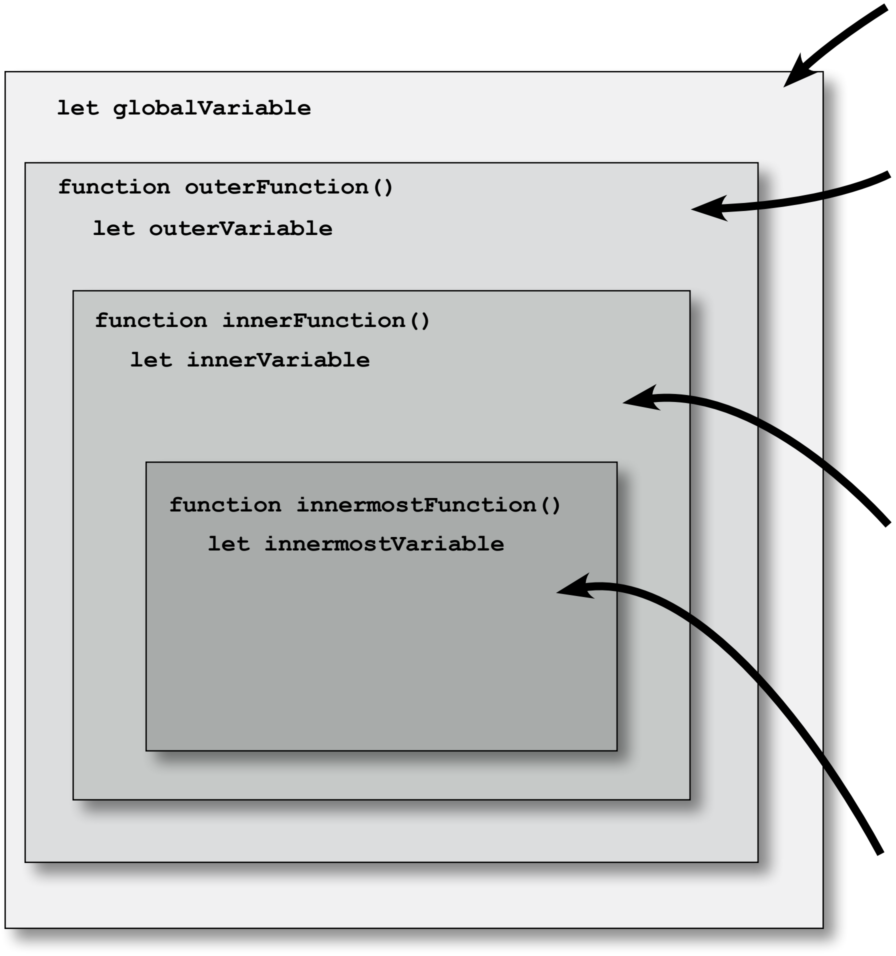
Alex: With lexical scope we can always tell the scope of a variable just by looking at the code. We don't have to look at any aspect of the runtime behavior of the code to figure out where a variable is defined and where it's not.

Sam: So here with these nested functions, if we look at a given variable, we just look up, or outward, until we find where that variable is defined?

Mara: I think so, but let's keep at it. I think we're finally getting on top of this JavaScript scope stuff.

A lexical scope refresher

Most languages, including JavaScript, use *lexical scope* to determine where variables are visible in code. As we've said, lexical scope is determined by looking at the structure of your code, rather than by runtime behavior. So, you can print out any JavaScript program and clearly identify the lexical scope of any given variable. With respect to functions, we can think of the lexical scope of variables like this: variables defined inside a function are accessible only within that function and any nested functions. Let's see what that looks like:


let globalVariable
function outerFunction()
let outerVariable
function innerFunction()
let innerVariable
function innermostFunction()
let innermostVariable

At the global level, only globalVariable is in scope. Variables nested in functions are not in scope.

outerFunction can see outerVariable and globalVariable. It cannot see innerVariable or innermostVariable because they are defined in nested functions, and outerFunction can't see into their scope.

innerFunction can see its own innerVariable and all the variables outward from it: outerVariable and globalVariable. It can't see into innermostFuction, so it can't see innermostVariable.

innermostFunction can see its own innermostVariable and all the variables outward from it: innerVariable, outerVariable, and globalVariable.

Variables defined inside a function are accessible only within that function and any nested functions.

Another look at our outer/inner functions

The same lexical scoping rules apply to nested blocks and nested functions. Let's revisit our outer and inner functions from the previous exercise to see how this works and why we encountered an error, remembering that to figure out the scope of a variable or function we look for the closest enclosing block where that variable or function is defined:

outerFunction is defined at the top level, so it's in the global scope and visible everywhere.

outerVariable is defined inside the outerFunction block, at the top, so it's visible everywhere in outerFunction, including any blocks inside outerFunction where it's not shadowed. In this example, we are not shadowing this variable anywhere.

```
function outerFunction() {

    let outerVariable = "I'm outside!";
```

innerFunction is defined in outerFunction's block, so it's visible everywhere in outerFunction, including any blocks nested inside outerFunction.

innerVariable is defined at the top of innerFunction, so it's visible everywhere in innerFunction, including any blocks inside innerFunction where it's not shadowed.

```
    function innerFunction() {

        let innerVariable = "I'm inside!";

        console.log(innerVariable);
```

What's the closest enclosing block where innerVariable is defined? innerFunction. Can we see innerVariable here? Yes. So we can print it.

```
        console.log(outerVariable);

    }
```

What's the closest enclosing block where outerVariable is defined? outerFunction. Can we see outerVariable here? Yes. So we can print it.

When we call outerFunction, this statement works fine.

```
    console.log(outerVariable);
```

What's the closest enclosing block where outerVariable is defined? outerFunction. Can we see outerVariable here? Yes. So we can print it.

Error!

```
    console.log(innerVariable);
```

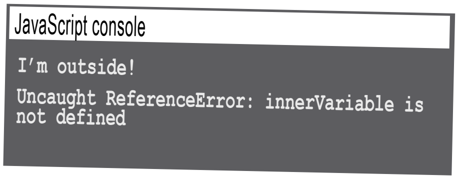
What's the closest enclosing block where innerVariable is defined? **Oh no! There isn't one.** It's not defined in the closest enclosing block, outerFunction, and it's not defined in the global scope.

```
    innerFunction();

}
```

The scope of innerFunction is the entire outerFunction block, so this function call should work, but we never get here.

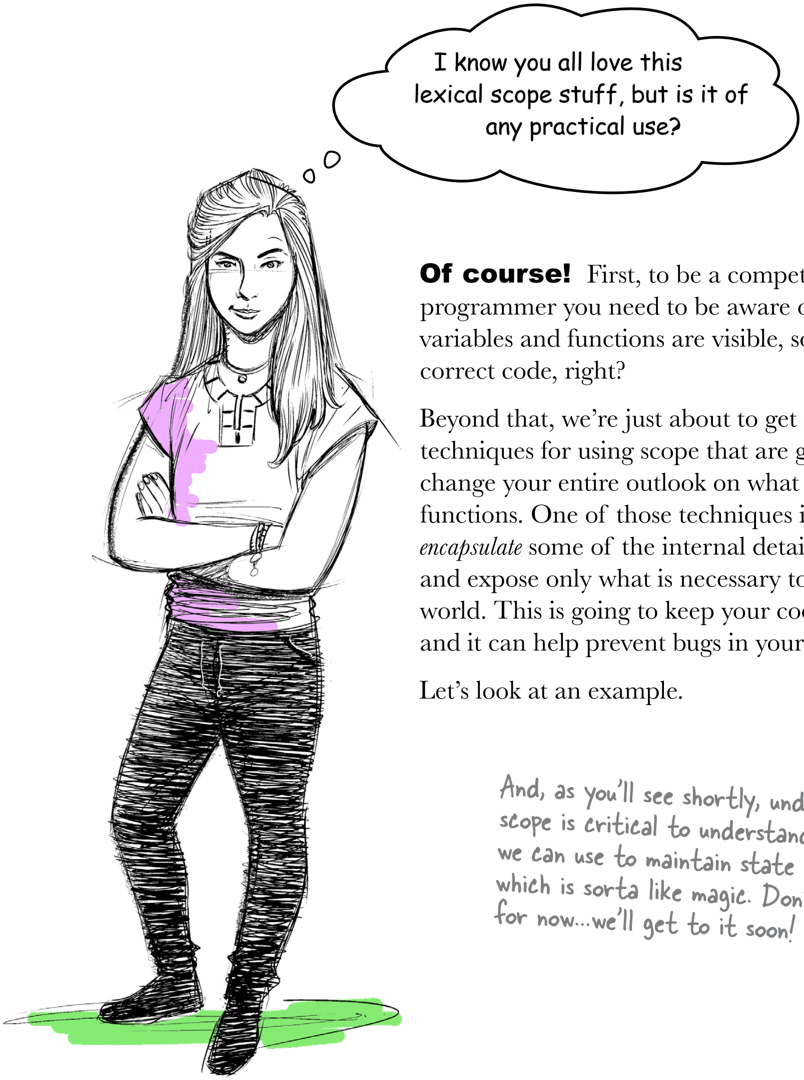
```
outerFunction();
```

Here, we call outerFunction and see this result because innerVariable is NOT defined.



JavaScript console

```
I'm outside!
Uncaught ReferenceError: innerVariable is
not defined
```


I know you all love this lexical scope stuff, but is it of any practical use?

Of course! First, to be a competent programmer you need to be aware of where your variables and functions are visible, so you can write correct code, right?

Beyond that, we're just about to get into some techniques for using scope that are going to change your entire outlook on what is possible with functions. One of those techniques is to protect or *encapsulate* some of the internal details of your code and expose only what is necessary to the outside world. This is going to keep your code organized, and it can help prevent bugs in your code too.

Let's look at an example.

And, as you'll see shortly, understanding lexical scope is critical to understandng closures, which we can use to maintain state beyond function calls, which is sorta like magic. Don't worry about this for now...we'll get to it soon!

Using scope for encapsulation

Sometimes, a little privacy can be a good thing. There are times when you want to limit what others can see. Take a fortune cookie, for example. The fortune is encapsulated inside the cookie, so only the person who gets a particular cookie gets to know what their future holds.

Let's see how we can use nested functions (meaning, nested scopes) to control who gets to see the fortunes hidden inside a bag of cookies.

Let's say you don't want other code peeking at your fortunes. All that code knows is that each time it calls fortuneCookie, it gets a fortune. All the details about the fortunes and the code to randomly produce a fortune are hidden inside the fortuneCookie function.

At this level of the code, can you iterate over the fortunes? No. Can you index into the fortunes array? No. You can't see the fortunes at all! The only way to get a fortune is to call the fortuneCookie function.

```
let fortune = fortuneCookie();
console.log(`Your Fortune: ${fortune}`);

function fortuneCookie() {
  function rand(max) {
    return Math.floor(Math.random() * max);
  }

  function getFortune() {
    const fortunes = [
      "Good things come to those who wait.",
      "A surprise is waiting for you around the corner.",
      "Embrace change; it brings new opportunities.",
      "Your hard work will soon pay off.",
      "A journey begins with a single step.",
    ];

    let i = rand(fortunes.length);
    return fortunes[i];
  }

  let fortune = getFortune();
  return fortune;
}
```

What does the rand function do? It encapsulates the process of producing a random number. We could change how we do this in the future, and it wouldn't affect the rest of the code at all. Cool.

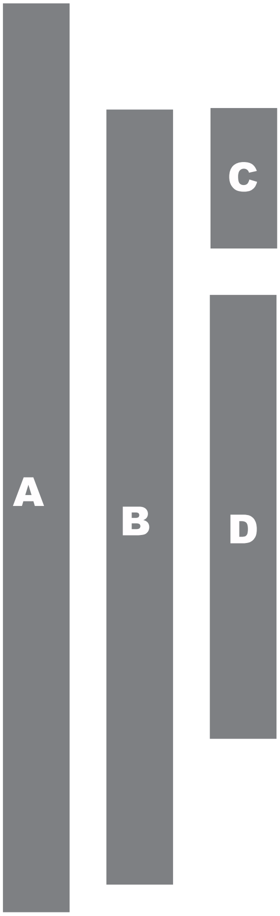
The getFortune function encapsulates the fortunes and the process of choosing a random fortune. Again, we could change the fortunes, or the process of selecting one for the cookie, and it wouldn't affect the rest of the code.

Here, we call getFortune to get a random fortune and return it as the result of calling fortuneCookie. All the details are encapsulated tidily by the two functions, rand and getFortune, making code maintenance a breeze.

Exercise

You’ve seen roughly what the fortune cookie code does. Now, go through and identify the scopes of the variables and functions, and write their corresponding scope letters in the spaces provided. We've done the first one for you. Check your answers at the end of the chapter.

Write your answsers here.



```
let fortune = fortuneCookie();
console.log(`Your Fortune: ${fortune}`);

function fortuneCookie() {
  function rand(max) {
    return Math.floor(Math.random() * max);
  }

  function getFortune() {
    const fortunes = [
      "Good things come to those who wait.",
      "A surprise is waiting for you around the corner.",
      "Embrace change; it brings new opportunities.",
      "Your hard work will soon pay off.",
      "A journey begins with a single step.",
    ];

    let i = rand(fortunes.length);
    return fortunes[i];
  }

  let fortune = getFortune();
  return fortune;
}
```

fortune: A

fortuneCookie: ____

rand: ____________

max: ____________

getFortune: ________

fortunes: _________

i: _______________

fortune: _________

Solution on page 538

Two important JavaScript scope rules

Here's a handy cheat sheet you can copy and pin up next to your computer while you're learning about scope in JavaScript. There are two rules you'll want to commit to memory. Here they are:

1 Within a function, if you want to determine where a variable is defined, search in its enclosing blocks (function or statement blocks), from the most nested to the least.

```
let a = 1;

function first() {
    let b = 2;
    let second = () => {
        let c = 3;
        return (a + b + c);
    };
    return (second());
}
console.log("Result: ", first());
```

We need the values of a, b, and c to know what sum to return.

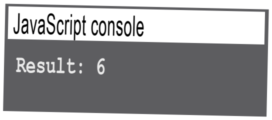
Here's how we find the values for a, b, and c.

a - Is a defined in second (the closest enclosing block)? No. Keep going. Is a defined in first? No, keep going. Is a defined globally? Yes! Found it.

b - Is b defined in second? No, keep going. Is b defined in first? Yes! Found it.

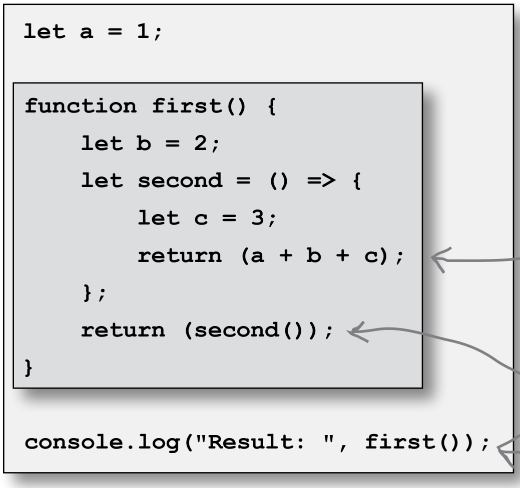
c - Is c defined in second? Yes! Found it.

Compute the result to return.



```
Result: 6
```

2 JavaScript functions are always evaluated in the same scoping environment in which they were defined.



```
let a = 1;

function first() {
    let b = 2;
    let second = () => {
        let c = 3;
        return (a + b + c);
    };
    return (second());
}

console.log("Result: ", first());
```

first is defined in the scoping environment that includes the global scope. In this environment, a and first are defined. This is the environment in which first is evaluated.

second is defined in the scoping environment that includes first and the global scope. In this environment, a, b, first, and second are defined. This is the environment in which second is evaluated.

there are no Dumb Questions

Q: When you say that lexical scope determines where a variable is defined, what do you mean?

A: This means that the program's structure determines the scope of the variables in it. Given a variable definition, you simply need to look at the block (function or statement) where the variable is defined, and that's its entire scope. If that block has nested blocks, the variable is visible to those nested blocks. Given a variable used in a function or block, look at the enclosing blocks to find its definition. If you can't find a variable's definition in an enclosing block, then it must be global or it's undefined.

Q: Just so I'm clear, a local scope is created by a function or a block, right?

A: That's right. The typical case is you have a function with its own code block. But statements like if and while often have blocks, and if you define variables within those blocks, they are local to those blocks.

Q: Do parameters get included in a function's local scope too?

A: Yes. As we've said before, you can consider parameters to be local variables in your functions, so they are included in the function's scope as well.

Q: What's a scoping environment? Is that the same as a scope?

A: You can think of a scope as being associated with a specific variable—as in, "What's the scope of a?" Think of a scoping environment as every variable and function that is defined in a given scope.

Q: Why is lexical scoping a good thing? I would have thought having everything defined in the global scope, so I can access every variable anywhere I want, might be better.

A: We can see how you might think that, but the advantage of lexical scope is that we can always look at the structure of the code to determine the scope that's in place when a variable is defined and when it's used, and figure out what its value should be from that.

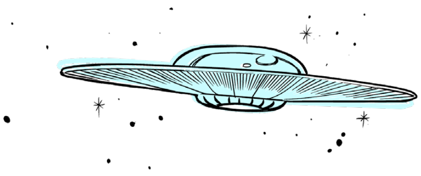
If every variable and function is accessible in the global scope, then you might end up with name conflicts, which can be a problem as applications get larger and multiple people are working on the code.

As we saw, lexical scope also allows us to encapsulate computation and data to control where it is visible. This can help to prevent unintended modifications by other code, and it helps us to keep our code organized and easier to maintain.

Given a variable, you simply need to look at the block where the variable is defined, and that's its entire scope.

JavaScript functions are always evaluated in the same scoping environment in which they were defined.

Let's make another change to our code



Okay, now that we have lexical scope under our belts, let's make another change to our outer and inner functions. Watch out, this one is a little mind-bending...

```
function outerFunction() {
    let outerVariable = "I'm outside!";

    function innerFunction() {
        let innerVariable = "I'm inside!";
        console.log(innerVariable);
        console.log(outerVariable);
    }

    console.log(outerVariable);
    return innerFunction;
}

let inner = outerFunction();
inner();
```

This part of the code is the same as before; nothing has changed.

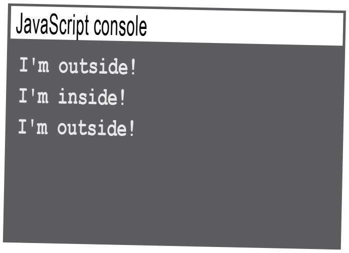
Check this out: we're no longer calling innerFunction from within outerFunction, we're RETURNING it.

And when we receive it from calling outerFunction, we immediately invoke it.

Brain Power

Think about everything you've learned about scope and functions. What is this code going to do?

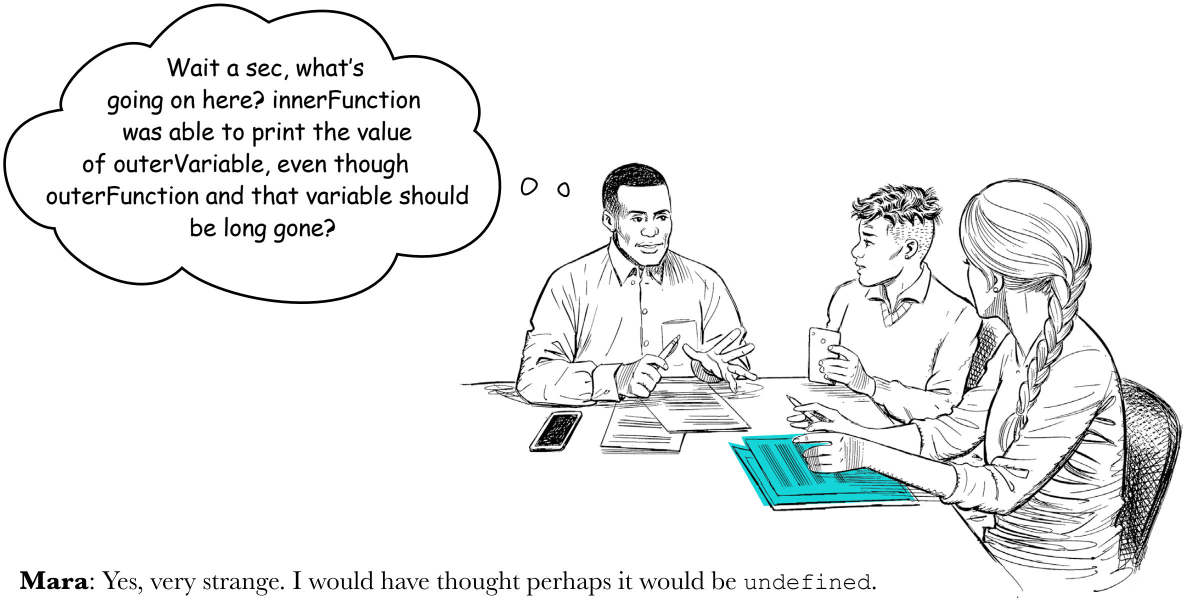
Would you believe it? How is that possible?



```
I'm outside!
I'm inside!
I'm outside!
```

Okay, this Brain Power has generated way more questions than answers.

How did innerFunction escape outerFunction and keep outerVariable alive?


Wait a sec, what's going on here? innerFunction was able to print the value of outerVariable, even though outerFunction and that variable should be long gone?

Mara: Yes, very strange. I would have thought perhaps it would be `undefined`.

Sam: Seriously...that's like defying the laws of physics or something. `outerVariable` doesn't even exist anymore…I mean, when a variable goes out of scope it ceases to exist. It's derezzed! Didn't you see *TRON*!?

Alex: Hold on, catch me up, I'm not following.

Sam: So, when we call `outerFunction`, the very last thing it does in its life is hand us back `innerFunction`. But at that point `outerFunction` is done, gone, along with all its variables.

Alex: Got it...and yet through some miracle `innerFunction` is still able to access the value of `outerVariable`, is that it?

Sam: You got it.

Mara: Theories?

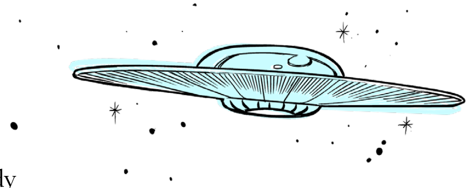
Sam: I don't know, is this just a fluke? Like, we run it a few more times and we start to get `undefined`?

Mara: I don't think so. I'm thinking about that lexical scope. Remember we said that functions are always evaluated in the same scope they're defined in.

Sam: So how on earth does that happen when the function that created the scope is gone? More science fiction! It's like we're on a different planet.

Mara: I don't know, we need to dig deep on this one.

Solving the mystery



Queue the *X-Files* theme song. How is it we're making use of a variable that was defined in a function within our lexical scope, but that has already completed its execution? As we've always understood it, local variables are created when a function starts and destroyed when a function completes. So how can we still be using this variable? Don't worry, the truth is out there.

And it's closer than you think: the reason this works is because of lexical scoping and a feature of JavaScript called a *closure*. In JavaScript, whenever a function is returned from another function, it is wrapped up with the function's lexical environment, meaning all the variables in its outer scope that it might need in the future. You can think of this as the function remembering its *environment*, even if the outer function has long since expired. And, as long as the inner function exists, those outer function variables will exist in memory for the inner function to make use of. That's a closure.

How to make a closure

Creating a closure is as simple as returning a function from a function:

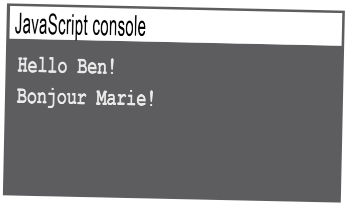
Here's a function that returns another function. createGreeting takes one argument, greeting.

```
function createGreeting(greeting) {
    return function(name) {
        return `${greeting} ${name}!`;
    };
}

let sayHello = createGreeting("Hello");
console.log(sayHello("Ben"));
let sayBonjour = createGreeting("Bonjour");
console.log(sayBonjour("Marie"));
```

It returns a function that is waiting on another argument, a name. When this function is executed, it will print the previously passed in greeting followed by the name.

Here's some test code; make sure you see how this code is operating before moving on.

Here's the output we got.



JavaScript console

```
Hello Ben!
Bonjour Marie!
```

Stepping through createGreeting

PRODUCTION: Head First
SCENE 2
TAKE 3
ROLL
SOUND
DATE
PRODUCTION CO. O'Reilly
DIRECTOR

Behind the Scenes

Let's take a closer look at how `createGreeting` is working:

```
function createGreeting(greeting) {
    return function(name) {
        return `${greeting} ${name}!`;
    };
}

let sayHello = createGreeting("Hello");
console.log(sayHello("Ben"));
```

---

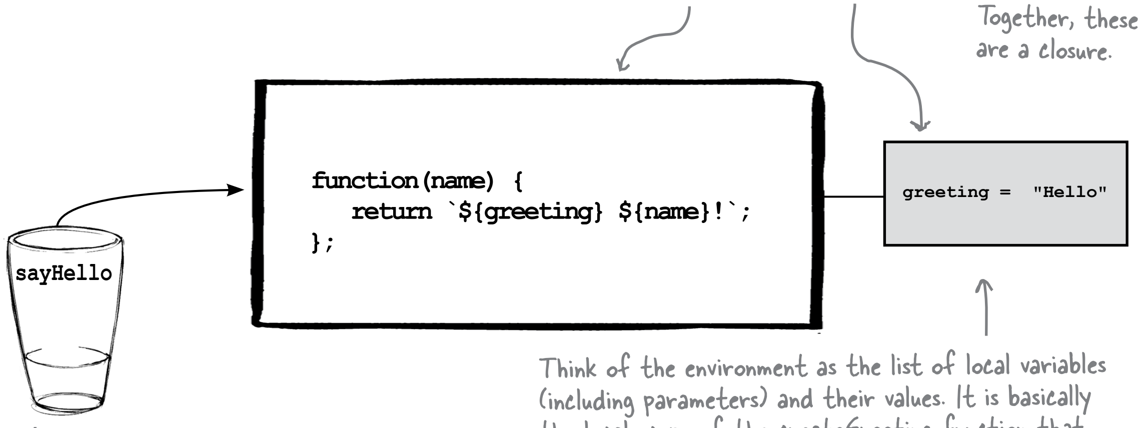
1. **First, we call createGreeting. We already know that createGreeting returns a function, so we create a sayHello variable and assign it that function. That function reference is linked to an environment that is the scope that existed when that function was created.**

```
let sayHello = createGreeting("Hello");
```

After this statement, we have a variable sayHello that refers to the function (plus an environment) returned from createGreeting.

The function, and its environment.

Together, these are a closure.



```
function(name) {
   return `${greeting} ${name}!`;
};
```

Think of the environment as the list of local variables (including parameters) and their values. It is basically the local scope of the createGreeting function that existed when the function we returned was created.

Our new variable.

PRODUCTION: Head First
SCENE 2
TAKE 3
ROLL
DATE
SOUND
PRODUCTION CO. O'Reilly
DIRECTOR

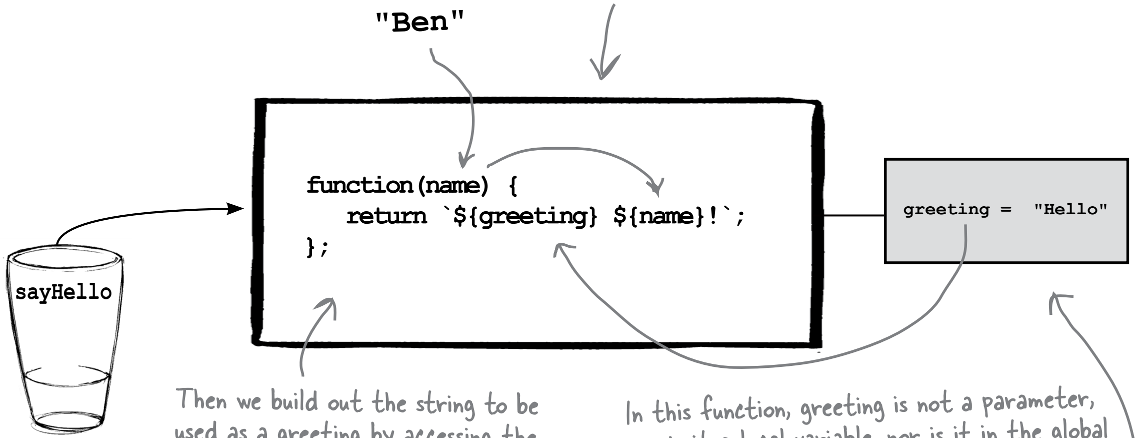
Behind the Scenes

2 **Next, we call sayHello with the argument "Ben". To do that we evaluate the code in the function's body, and we do that in the context of the function's environment, like this:**

```
console.log(sayHello("Ben"));
```

We're calling sayHello with the argument "Ben" (and whatever we get back we'll print to the console).

To evaluate this, the string "Ben" is first passed into the name parameter.



```
function(name) {
   return `${greeting} ${name}!`;
};
```

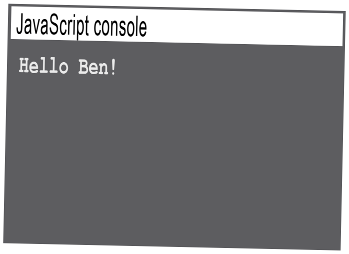
Then we build out the string to be used as a greeting by accessing the name parameter plus the greeting variable, which is available in the local environment of the closure.

In this function, greeting is not a parameter, nor is it a local variable, nor is it in the global scope where we're calling sayHello("Ben"), so the only other place it can be is in the environment associated with the function. If it's not there, it's undefined. But in our case it is there, so we use it.

3 **Last, we return the string constructed from the name parameter and the greeting variable stored in the environment. The string is then displayed in the console.**

```
console.log(sayHello("Ben"));
```

The sayHello function is a closure—a function plus an environment—so it can be called over and over with a different name argument each time. It will use the "Hello" stored in the closure's environment to produce the greeting.



```
Hello Ben!
```

Sharpen your pencil

We've modified the fortune cookie example from earlier in the chapter to create a closure. Here's your task:

(1) Write in what code is in the function returned by createFortuneCookies and stored in getFortune.

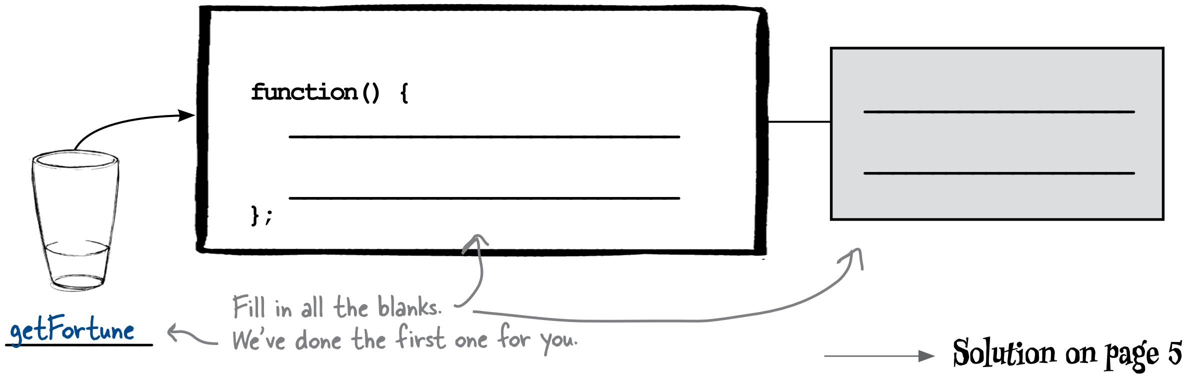
(2) Write in the variables that are in the environment in the closure.

Check your answers at the end of the chapter before moving on, and run the code to see if it does what you expect.

```
function createFortuneCookies() {
    const fortunes = [
        "Good things come to those who wait.",
        "A surprise is waiting for you around the corner.",
        "Embrace change; it brings new opportunities.",
        "Your hard work will soon pay off.",
        "A journey begins with a single step.",
    ];
    function rand(max) {
        return Math.floor(Math.random() * max);
    }
    function getAFortune() {
        let i = rand(fortunes.length);
        return fortunes[i];
    }
    return getAFortune;
}
// Make a bag of fortuneCookies
let getFortune = createFortuneCookies();
// Pull one out and open it!
console.log(`Your Fortune: ${ getFortune() }`);
```

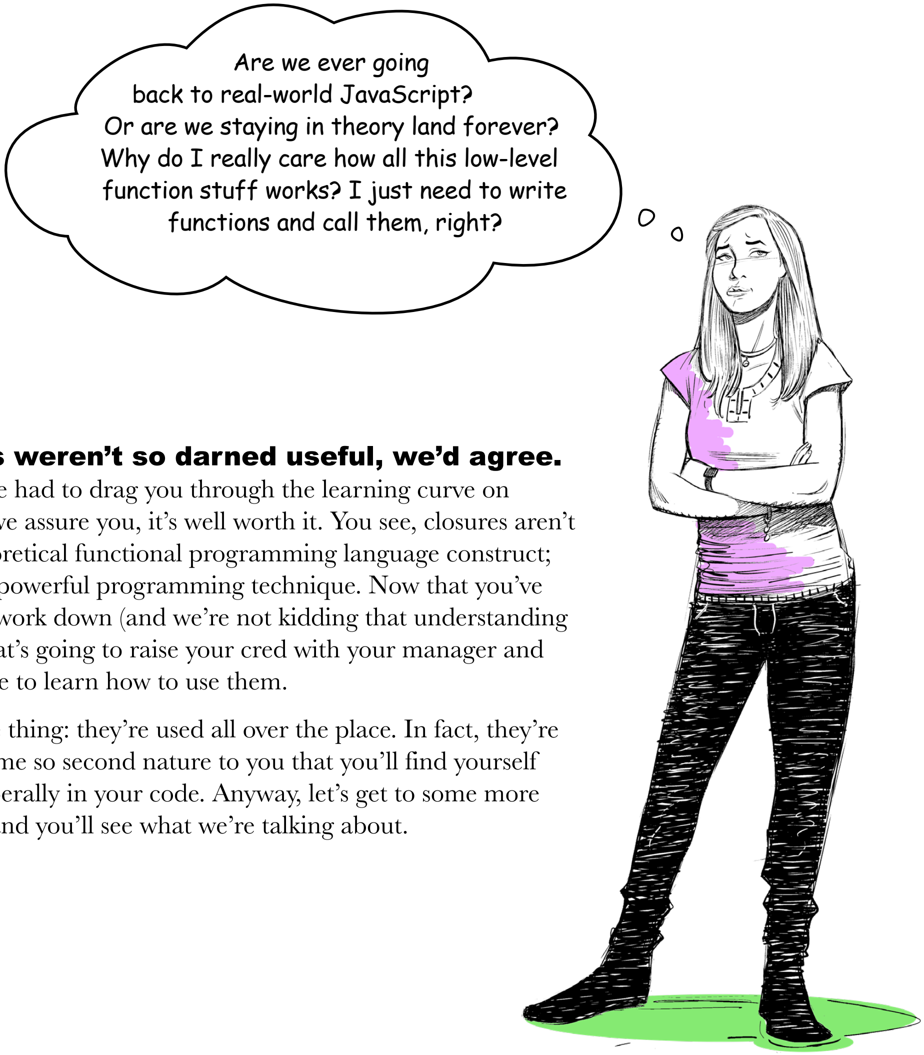
Hint: look for variables used by the function we return to getFortune that aren't local to that function, but are included in the scope when the function is created.

Your job is to fill in the blanks in the closure created by this code.



Fill in all the blanks. We've done the first one for you.

Solution on page 539


Are we ever going back to real-world JavaScript? Or are we staying in theory land forever? Why do I really care how all this low-level function stuff works? I just need to write functions and call them, right?

If closures weren't so darned useful, we'd agree.

We're sorry we had to drag you through the learning curve on closures, but we assure you, it's well worth it. You see, closures aren't just some theoretical functional programming language construct; they're also a powerful programming technique. Now that you've got how they work down (and we're not kidding that understanding closures is what's going to raise your cred with your manager and peers), it's time to learn how to use them.

And here's the thing: they're used all over the place. In fact, they're going to become so second nature to you that you'll find yourself using them liberally in your code. Anyway, let's get to some more closure code and you'll see what we're talking about.

Using closures to implement a magic counter

Ever think of implementing a counter function? It usually goes like this:

```
let count = 0;

function counter() {
    count = count + 1;
    return count;
}
```

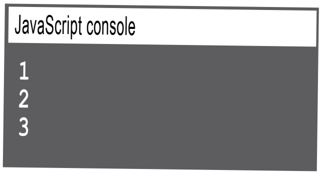
We have a global variable, count.

Each time we call counter, we increment the global count variable and return the new value.

And we can use our counter like this:

```
console.log(counter());
console.log(counter());
console.log(counter());
```

So, we can count and display the value of the counter like this.


JavaScript console

```
1
2
3
```

The only issue with this is that we have to use a global variable for `count`, which can be problematic if you're developing code with a team (because people often use the same names, which end up clashing).

What if we were to tell you there is a way to implement a counter with a totally local and protected `count` variable? That way, you'll have a counter that no other code can ever clash with, and the only way to increment the counter value will be through the function (otherwise known as a closure).

To implement this with a closure, we can reuse most of the code above. Watch and be amazed:

```
function makeCounter() {
    let count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
```

Here, we're putting the count variable in the function makeCounter. So now count is a local variable, not a global variable.

Now, we create the counter function, which increments the count variable...

...and return the counter function.

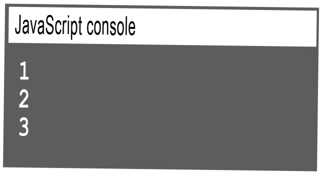
This is the closure. It holds count in its environment.

Think this magic trick will work? Let's try it and see…

Test drive your magic counter

We added a bit of code to test the counter. Give it a try!

```
function makeCounter() {
    let count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
let doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```



```
1
2
3
```

Our counter works...we get solid counting results.

Looking behind the curtain...

Let's step through the code to see how the counter works:

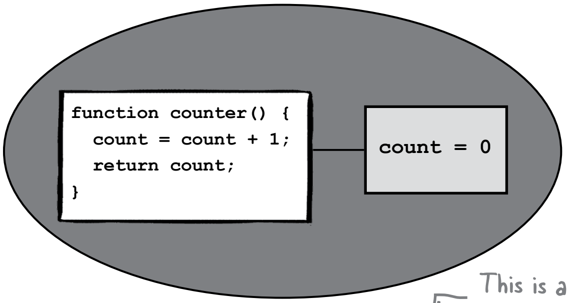
① We call makeCounter, which creates a counter function and returns it along with an environment containing the variable count. In other words, it creates a closure. The function returned from makeCounter is stored in doCount.

② We call the function doCount. This executes the body of the counter function.

③ When we encounter the variable count, we notice it's not a local variable, so we look it up in the environment and retrieve its value. We increment count, save the new value back into the environment, and return that new value to where doCount was called.

④ We repeat steps 2 and 3 each time we call doCount.

When we call doCount (which is a reference to counter) and need to get the value of count, we use the count variable that's in the closure's environment. The outside world (the global scope) never sees the variable count, but we can use it anytime we call doCount. And there's no other way to get to count except by calling doCount.

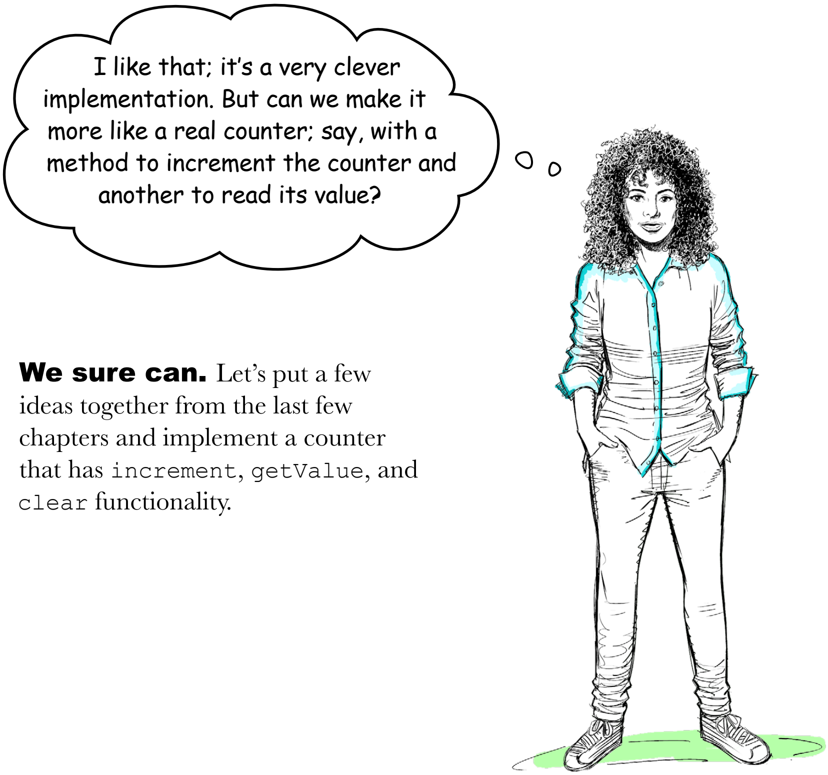


```
function counter() {
  count = count + 1;
  return count;
}
```

This is a closure.

```
function makeCounter() {
    let count = 0;

    function counter() {
  ③    count = count + 1;
        return count;
    }
    return counter;
}
① let doCount = makeCounter();
② console.log(doCount());
   console.log(doCount());
④ console.log(doCount());
```

When we call makeCounter, we get back a closure: a function with an environment.


I like that; it's a very clever implementation. But can we make it more like a real counter; say, with a method to increment the counter and another to read its value?

We sure can. Let's put a few ideas together from the last few chapters and implement a counter that has `increment`, `getValue`, and `clear` functionality.

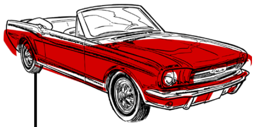
Implementing a counter with a closure

Okay, here's a counter with a little more functionality. Let's see how it works:

```
function createCounter() {
    let count = 0;

    return {
        increment() {
            count += 1;
        },
        getValue() {
            return count;
        },
        clear() {
            count = 0;
        }
    };
}
```

Just like last time, we have our count variable in the outer function; this is going to belong to the environment of any functions (or methods!) returned.

Only now we have an object literal with three methods being returned. These methods are each going to have count in their lexical environment, meaning in their closure.

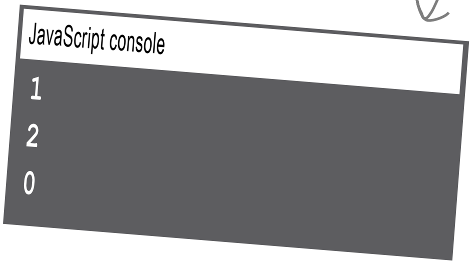


Test Drive

Let's give it a test drive. Here's our test code:

```
let counter = createCounter();
counter.increment();
console.log(counter.getValue());
counter.increment();
console.log(counter.getValue());
counter.clear();
console.log(counter.getValue());
```

Our counter works!



```
JavaScript console
1
2
0
```

Exercise

Write a function, secretMaker, that takes a password as an argument and returns a function that, when called, tests to see if you guessed the correct password. If you guessed correctly it returns true and otherwise it returns false. Like this:

```
let secret = secretMaker("wizard");
secret("hi");         // returns false
secret("wizard");     // returns true
secret("bye");        // returns false
```

Your code should run like this.

Solution on page 536

there are no Dumb Questions

Q: Do I need to understand how the environment works in detail?

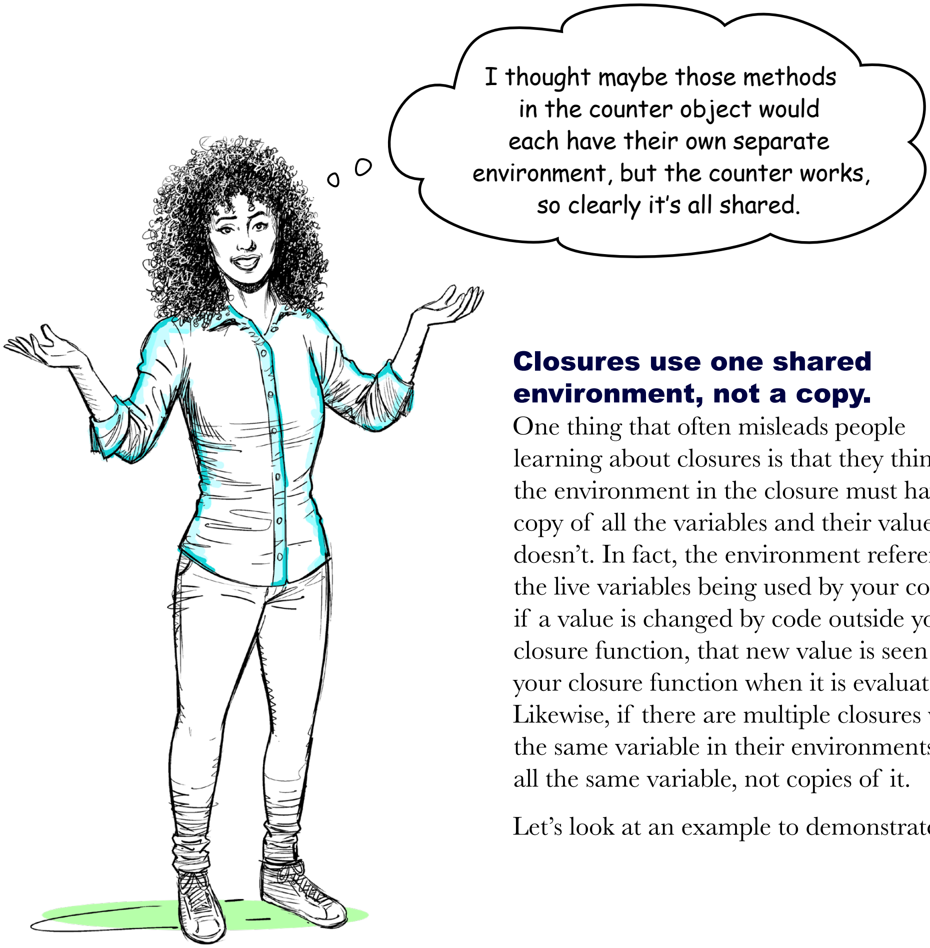
A: No. What you need to understand are the lexical scoping rules for JavaScript variables, and we've covered those. But now you know that if you have a function that is returned from within a function, it carries its environment around with it as part of a closure.

Q: Do all functions come with an environment, or just ones returned from another function?

A: All functions have an environment, so technically, all functions are part of a closure. It's just that many times, that closure is empty. We're looking at specific examples of when the closure contains varibles and their values.

Q: Is there a name for the variables included in the environment?

A: Yes, the variables in the environment are called "free variables" because they weren't defined in the local scope of the function that was returned, they were defined in the enclosing scope. So, count was a free variable in counter (i.e., not locally defined), but it was needed to execute the doCount function, and that's why it's included in the closure.


I thought maybe those methods in the counter object would each have their own separate environment, but the counter works, so clearly it's all shared.

Closures use one shared environment, not a copy.

One thing that often misleads people learning about closures is that they think the environment in the closure must have a copy of all the variables and their values. It doesn't. In fact, the environment references the live variables being used by your code, so if a value is changed by code outside your closure function, that new value is seen by your closure function when it is evaluated. Likewise, if there are multiple closures with the same variable in their environments, it's all the same variable, not copies of it.

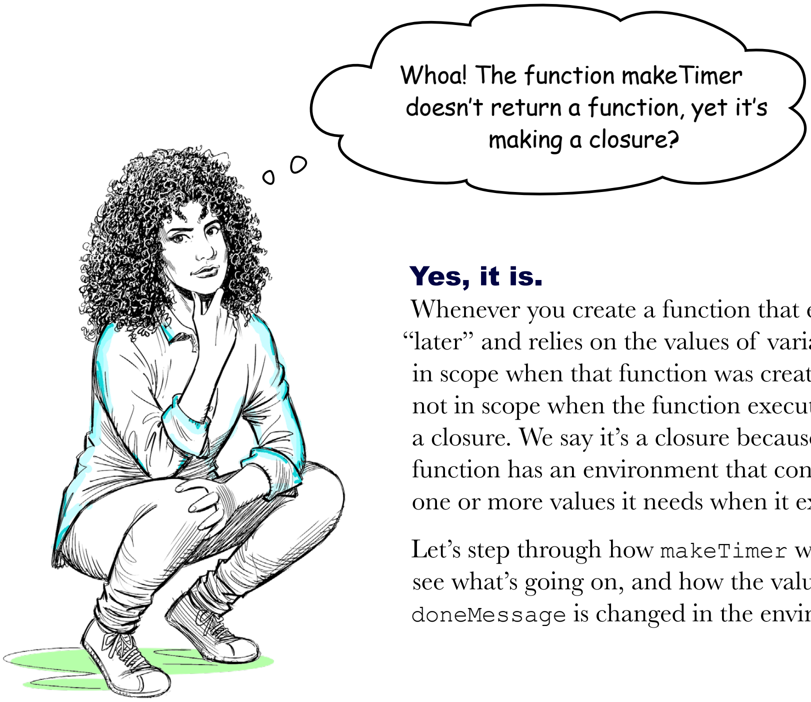
Let's look at an example to demonstrate this.

Here's a function, makeTimer, that creates a closure. Can you see how the anonymous function in setTimeout is creating a closure? And what variables is it expeecting in the environment? How about doneMessage?

```
function makeTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "OUCH!";
}
makeTimer("Cookies are done!", 5000);
```

The closure is created here.

Now we're changing the value of doneMessage after we call setTimeout.


Whoa! The function makeTimer doesn't return a function, yet it's making a closure?

Yes, it is.

Whenever you create a function that executes "later" and relies on the values of variables in scope when that function was created but not in scope when the function executes, it's a closure. We say it's a closure because that function has an environment that contains one or more values it needs when it executes.

Let's step through how `makeTimer` works to see what's going on, and how the value of `doneMessage` is changed in the environment.

Even though we're not returning a function from makeTimer, we're creating a function that will run later (5 seconds later, to be precise). That function lives on beyond the execution (and scope) of makeTimer, AND it requires the value of doneMessage when it runs, which we change in the one shared environment. Let's see how this works...

```
function makeTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "OUCH!";
}
makeTimer("Cookies are done!", 5000);
```

How makeTimer works

1. After defining the function makeTimer, we call makeTimer with a string for doneMessage and a value of 5000 for n:

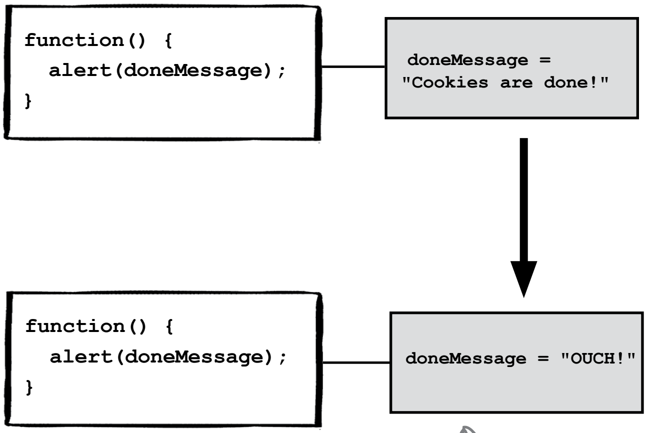
```
function makeTimer(doneMessage, n) {
    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "OUCH!";
}
makeTimer("Cookies are done!", 5000);
```

First, makeTimer is defined.

Next, we call makeTimer, and makeTimer starts executing.

2. We call setTimeout and pass to it a function expression that will need the variable doneMessage when it runs. A closure is created containing the function along with a reference to the environment.

```
setTimeout(function() {
    alert(doneMessage);
}, n);
```



3. We change the value of doneMessage to "OUCH!" in code outside of the closure. That changes doneMessage in the **same environment** that is used by the closure.

```
doneMessage = "OUCH!";
```

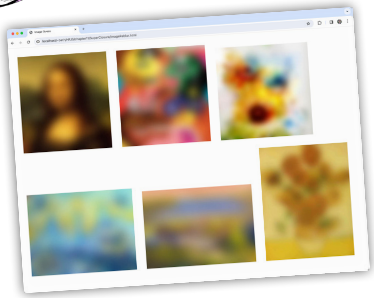
4. makeTimer finishes up and its scope disappears. 5,000 milliseconds (5 seconds) later, the function in the closure is called. This function references the doneMessage variable, which is now set to "OUCH!" in the environment, so we see "OUCH!" in the alert.

```
function() { alert(doneMessage); }
```

When the function is called, it uses the value for doneMessage that's in the environment, which is the new value we set it to earlier, in makeTimer.

Remember that JavaScript functions are always evaluated in the environment in which they were defined.

Super Closure Exercise



Revisit this code from Chapter 9. Recall, this is our image game code to unblur an image when you click on it, and reblur it a few seconds later. Can you modify the code to use a closure, and eliminate the need for the third argument to setTimeout? Check our solution at the end of the chapter before you go on.

```
window.onload = init;
function init() {
    let images = document.getElementsByTagName("img");
    for (let i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};

function showAnswer(eventObj) {
    let image = eventObj.target;
    let name = image.id;
    image.src = `${name}.jpg`;

    setTimeout(reblur, 2000, image);
}

function reblur(image) {
    let name = image.id;
    image.src = `${name}blur.jpg`;
}
```

Hint: think carefully about the image variable here. Draw out the environment created by setTimeout and think about where the image variable will come from when the function you pass to setTimeout is called.

2nd hint: Might you eliminate a function in this process?

→ Solution on page 540

Mara: So, I was sort of bragging to Initech that we really have our lexical scope down with JavaScript.

Sam: Oh yeah? I suppose they started peppering you with lexical scope interview questions?

Mara: They gave us a challenge, and if we can do it by the end of the day, dinner is on them.

Sam: And if we can't?

Mara: Well, then it's on us.

Alex: Thanks, Mara! I guess you'd better fill us in on the challenge. How bad is this going to be?

Mara: The challenge is to write a function that takes another function as an argument, and only lets that function be invoked *once ever*.

Alex: That's like some sort of weird Singleton object-oriented pattern, but it's a function.

Mara: Sort of. But how do we do that? How can we keep a function from being called more than once?

Sam: I'm pretty sure closures are going to save us here.

Mara: Good thinking. If we can keep some local state that keeps track of how many times the function has been called, we can make sure it isn't invoked again after it's been invoked once.

Sam: I think so...let's noodle this out with some pseudocode.

Sharpen your pencil

Think through how you would capture both the function (that can only be called once) and the state needed to track its calls in a closure. What state do you need? What logic do you need to write to ensure it's only called one time? Write down some ideas and pseudocode for a solution.

We got a head start; we went ahead and added a function called onlyOnceMaker that will take a function and turn it into a function that can only be called once.

```
function onlyOnceMaker(_______) {

    function run() {




    }


    return _________________;
}
```

And we added an inner function where all the logic will live.

____

____

____

____

How would you test it?

→ Solution on page 541

Sam: Alright, dinner is on them tonight!

Mara: I hope so, let's work through this.

Sam: So how exactly is this going to work? You said the challenge was to write a function that takes another function as an argument and only lets it be invoked once ever.

Mara: Right, so given any function, we get a new one back that can only be invoked one time.

Alex: It's like a one-time-use ticket. Once you call the function once, that's it. But what happens if you call it a second, or third, time?

Sam: Then it just does nothing. I mean, you can still call it, but it doesn't do anything.

Mara: Right. So let's set it up like this: we create a function, say, `onlyOnceMaker`, that accepts another function, let's call it `func`, that we'd like to only be invoked once.

Alex: Okay, then what?

Mara: Then we need to return a function that is going to be the actual function folks are trying to invoke. It's a version of the `func` we pass in, a version that can only be called once.

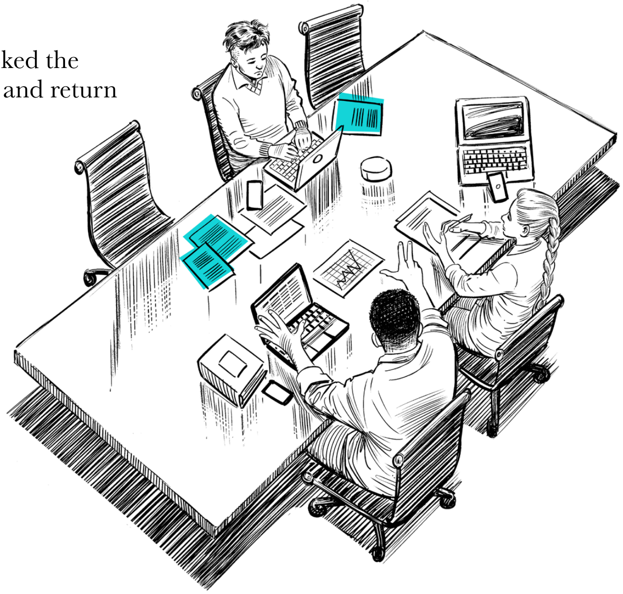
Alex: Ah, so we're returning a new function that does what the original does, but only once.

Mara: You're getting it.

Sam: To implement this we're going to need some state to track if `func` has been called before. Maybe a simple `count` variable. We did the same kind of thing with the `counter` code.

Mara: Exactly. And if the `count` is less than 1, we haven't invoked the `func` function yet, so we can call it. Otherwise, we just ignore it and return from the function.

Sam: Nice! Shall we try to code this?



Implementing onlyOnceMaker

Enough talk. Time to try to implement the `onceOnlyMaker` function. You've done the training; you've got this:

```
function onlyOnceMaker(func) {
    let calls = 0;

    function run() {
        if (calls < 1) {
            func();
        }
        calls++;
    }
    return run;
}
```

onlyOnceMaker takes a function.

It also has a local variable to track the number of times the function has been called.

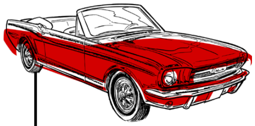
Here's our closure, run, which will be used to attempt to invoke the func function.

If it has been invoked before, we'll just quietly not invoke func; otherwise, we'll invoke it.

Here we track the number of calls to the function.

And here we return the closure.

Trace this code before you look at the output in the Test Drive below. Does this look right to you?



Test Drive

Here's our output. Is dinner on Initech tonight?

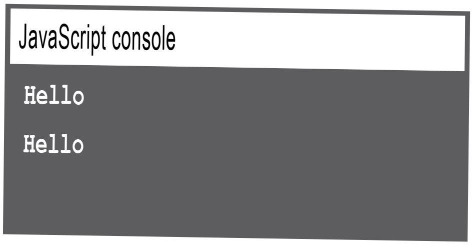
```
function sayHello() {
    console.log("Hello");
}

let helloOnce = onlyOnceMaker(sayHello);


helloOnce();
helloOnce();
sayHello();
```

Should print "Hello".

This is the second call; this should NOT print "Hello".

This is the original function; it still prints "Hello".

Here's our output. The first "Hello" is from the first call to our helloOnce. The second call doesn't print anything, and then we test the original function, which of course still prints "Hello".



JavaScript console

```
Hello
Hello
```

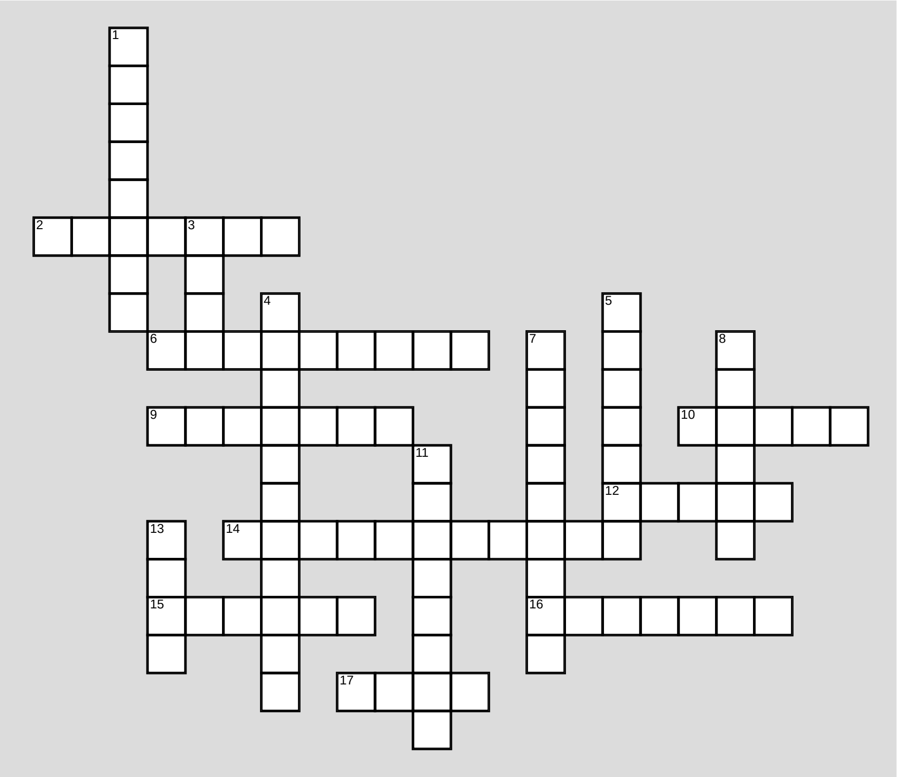
We've totally nailed scope and closures!
I feel like we got some serious JavaScript superpowers in this chapter.
Get ready for dinner on Initech!

Bullet Points

- **Default parameters** are default values assigned to parameters if the corresponding arguments are missing.
- **Rest parameters** use the rest operator, ... (three dots), to bundle up all the arguments passed to a function into one array.
- The **spread operator**, also three dots (...) allows you to spread out an object or array into function parameters, or into another object or array.
- Function **hoisting** refers to the first pass the browser's JavaScript interpreter takes over your code to find all the function declarations and define them, before it does a second pass to execute your code.
- Function expressions (including arrow functions) assigned to variables are not hoisted, so they are evaluated and executed in the second pass.
- **Nested functions** are functions defined inside another function.
- A nested function has local scope, just like other local variables.
- **Lexical scope** means that we can determine the scope of a variable by reading the structure of our code.
- **Scope** determines where your variables and functions are visible.
- You can use scope to **encapsulate** data and computation to limit its visibility and accessibility to code outside a function.
- Two important rules of scope are how to determine where a variable is defined by searching enclosing blocks, and that functions are evaluated in the environment in which they were defined.
- A **closure** is a function linked with an environment.
- A closure captures the values of variables in scope at the time the function is defined and the closure is created.
- If you execute a function closure in a different context in which it was created, the values of **free variables** are determined by the closure's environment.
- Closures are usually created by returning a function from a function, or passing a function to another function (i.e., with higher-order functions).

JavaScript cross

Time for another crossword puzzle to burn some JavaScript into those neuron pathways.



ACROSS

2. Function declarations are _______ on the browser's first pass through your code.

6. An _______ function is a function expression that has no name.

9. _______ scope means you can understand the scope of your variables by reading the structure of your code.

10. Parameters are _______ variables, so they're included in the environment where variables are defined.

12. A function declaration nested in another function has _______ scope.

14. An _______ provides values for free variables in a function.

15. The _______ operator spreads an array or object into arguments or other arrays or objects.

16. We often use setTimeout to create a timer for making _______.

17. A _______ variable is one that's not defined in the local scope.

DOWN

1. _______ expressions assigned to variables are not hoisted.

3. Movie the word "derezzed" was used in.

4. A function with an _______ attached to it is called a closure.

5. _______ values mean you can sometimes leave out an argument.

7. A function expression evaluates to a function _______.

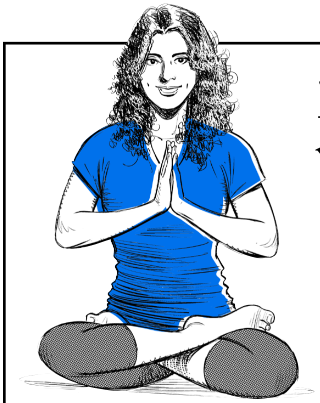
8. A function expression assigned to a variable at the top level of your code has _______ scope.

11. To get a raise, you should understand how _______ work.

13. The _______ operator can take multiple arguments and put them in one parameter.

→ Solution on page 542

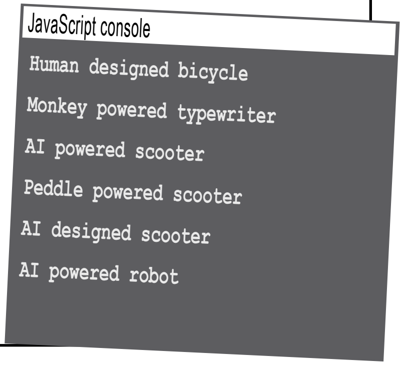
From page 493



BE the Browser Solution

Your job is to act like you're the browser. You need to evaluate the code and produce its output. Here's our solution.

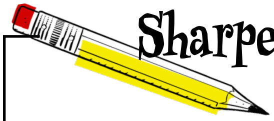
```
function pitch(noun="AI", verb="powered", obj="scooter") {
    console.log(noun, verb, obj);
}
pitch("Human", "designed", "bicycle");
pitch("Monkey", undefined, "typewriter");
pitch();
pitch("Peddle");
pitch(undefined, "designed");
pitch(undefined, undefined, "robot");
```



JavaScript console

```
Human designed bicycle
Monkey powered typewriter
AI powered scooter
Peddle powered scooter
AI designed scooter
AI powered robot
```

From page 494



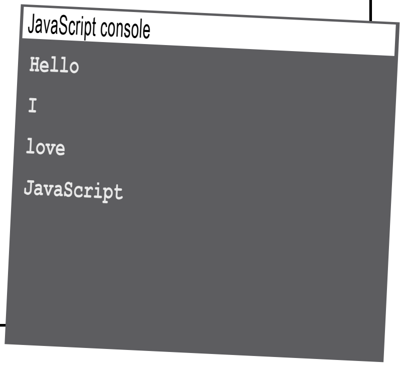
Sharpen your pencil Solution

Write a function named echo that takes any number of strings and prints out each one to the console log. Here's our solution.

```
function echo(...messages) {
    messages.forEach(m => console.log(m));
}
```

Here we're using a rest parameter and iterating over the array to print all its items.

```
echo("Hello", "I", "love", "JavaScript");
```



JavaScript console

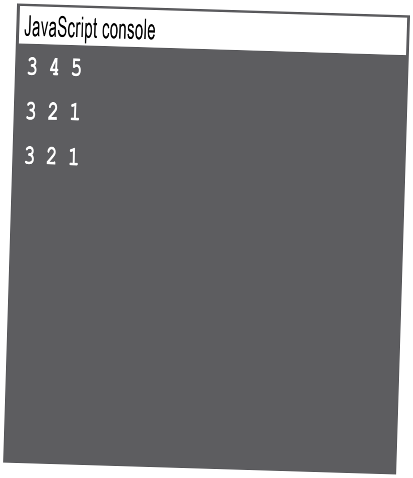
```
Hello
I
love
JavaScript
```

Sharpen your pencil Solution

From page 496

It's your turn. Test your function syntax chops and figure out what this code prints to the console. Write your answers in the space below. Here's our solution.

```
let arr = [1, 2, 3, 4, 5];

function foo(a = 3, b = 2, c = 1) {
    output = `${a} ${b} ${c}`;
    console.log(output);
}

function bar(a, b, ...c) {
    foo(...c);
}

bar(...arr);
bar(1, 2);
bar();
```



JavaScript console

```
3 4 5
3 2 1
3 2 1
```

Exercise Solution

From page 524

Write a function secretMaker. It takes a password as an argument and returns a function to test if you guessed the correct password. If you guessed correctly it returns true, otherwise false. Like this:

```
let secret = secretMaker("wizard");
secret("hi");       // returns false
secret("wizard");   // returns true
secret("bye");      // returns false
```

Your code should run like this.

Here's our solution:

```
function secretMaker(password) {
    return function(guess) {
        return (password === guess);
    }
}
```

The password variable and its value are stored in the closure of the function returned from secretMaker.

From page 502

BE the Browser Solution

Below, you'll find JavaScript code. Your job is to play like you're the browser evaluating the code. In the space to the right, record each function as it gets created. Remember to make two passes over the code: the pass that processes declarations, and the second pass that handles expressions.

Write, in order, the names of the functions as they are created. If a function is created with a function expression put the name of the variable it is assigned to. Here's our solution.

```
let midi = true;
let type = "piano";
let midiInterface;

function play(sequence) {
    // code here
}
let pause = function() {
    stop();
}
function stop() {
    // code here
}

function createMidi() {
   // code here
}

if (midi) {
    midiInterface = (type) => {
        // code here
    };
}
```

play

stop

createMidi

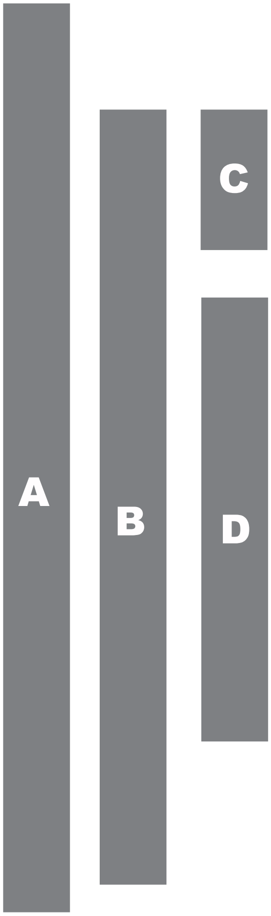
pause

midiInterface

Exercise Solution

From page 511

You've seen roughly what the fortune cookie code does. Now, go through and identify the scopes of the variables and functions, and write their corresponding scope letters in the spaces provided. Here's our solution.



```
let fortune = fortuneCookie();
console.log(`Your Fortune: ${fortune}`);

function fortuneCookie() {
  function rand(max) {
    return Math.floor(Math.random() * max);
  }

  function getFortune() {
    const fortunes = [
      "Good things come to those who wait.",
      "A surprise is waiting for you around the corner.",
      "Embrace change; it brings new opportunities.",
      "Your hard work will soon pay off.",
      "A journey begins with a single step.",
    ];

    let i = rand(fortunes.length);
    return fortunes[i];
  }

  let fortune = getFortune();
  return fortune;
}
```

fortune: A

fortuneCookie: A

rand: B

max: C

getFortune: B

fortunes: D

We can't see fortunes in any scope but D.

i: D

fortune: B

Notice that the fortune in fortuneCookie (B) is shadowing the fortune in the global scope (A).

From page 519

Sharpen your pencil Solution

We’ve modified the fortune cookie example from earlier in the chapter to create a closure. Here’s your task:

(1) Write in what code is in the function returned by createFortuneCookies and stored in getFortune.

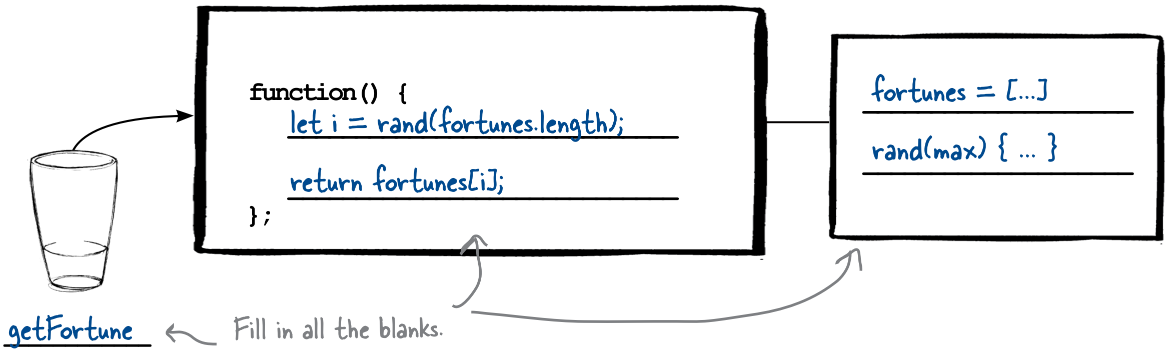
(2) Write in the variables that are in the environment in the closure.

Here’s our solution.

```
function createFortuneCookies() {
    const fortunes = [
        "Good things come to those who wait.",
        "A surprise is waiting for you around the corner.",
        "Embrace change; it brings new opportunities.",
        "Your hard work will soon pay off.",
        "A journey begins with a single step.",
    ];
    function rand(max) {
        return Math.floor(Math.random() * max);
    }
    function getAFortune() {
        let i = rand(fortunes.length);
        return fortunes[i];
    }
    return getAFortune;
}
// Make a bag of fortuneCookies
let getFortune = createFortuneCookies();
// Pull one out and open it!
console.log(`Your Fortune: ${ getFortune() }`);
```

The two variables not defined in getAFortune required to execute it after we return it to getFortune are the rand function and the fortunes array.

Your job is to fill in the blanks in the closure created by this code.



```
function() {
    let i = rand(fortunes.length);
    return fortunes[i];
};
```

fortunes = [...]

rand(max) { ... }

getFortune

Fill in all the blanks.

Super Closure Exercise Solution

From page 528

Revisit this code from Chapter 9. Recall, this is our image game code to unblur an image when you click on it, and reblur it a few seconds later. Can you modify the code to use a closure, and eliminate the need for the third argument to setTimeout? Here's our solution.

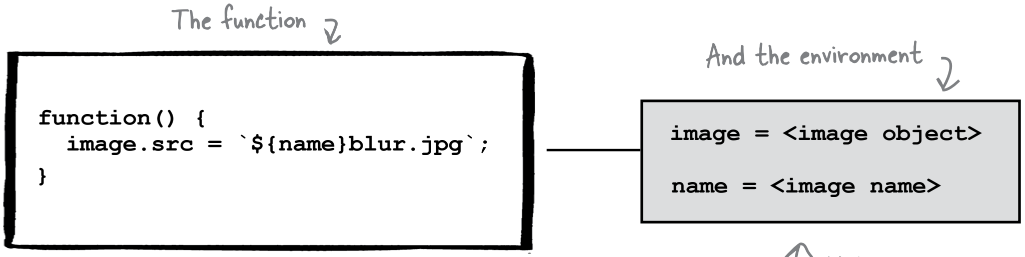
```
window.onload = function() {
    let images = document.getElementsByTagName("img");
    for (let i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
}

function showAnswer(eventObj) {
    let image = eventObj.target;
    let name = image.id;
    image.src = `${name}.jpg`;

    setTimeout(function() {
        image.src = `${name}blur.jpg`;
    }, 2000);
}
```

This part stays essentially just the same, except we made it slightly more concise by turning init into an anonymous function expression assigned to the window's onload property (just for extra credit, you know).

Here's where the magic happens. image and name are defined in showAnswer, which means they're defined in the scoping environment when we create the anonymous function expression that replaces reblur.

As you already know, passing a function that refers to variables in scope when the function is created creates a closure with an environment containing those variables. So, when this function is executed in 2 seconds, it will find both image and name in the environment linked with the closure.

And here's what the closure looks like...



The function

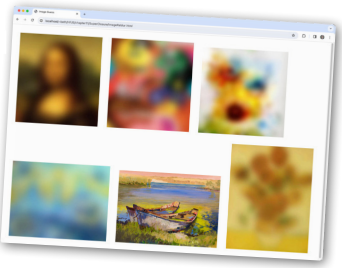
```
function() {
  image.src = `${name}blur.jpg`;
}
```

And the environment

```
image = <image object>

name = <image name>
```

Note that image will be the image object you clicked on to unblur it, and name will be the string name of the image (like "zero", "one", etc.).

Test Drive



One last test drive to make sure the image game with a closure works...and it does!

Nice work.

Sharpen your pencil Solution

From page 530

Think through how you would capture both the function (that can only be called once) and the state needed to track its calls in a closure. What state do you need? What logic do you need to write to ensure it's only called one time? Here's our pseudocode.

We got a head start; we went ahead and added a function called onlyOnceMaker that will take a function and turn it into a function that can only be called once.

```
function onlyOnceMaker( func ) {
    set counter to 0

    function run() {

        if counter is less than 1 then invoke
        the func function

        otherwise, just return

    }

    return run;
}
```

And we added an inner function where all the logic will live.

```
function sayHello {console.log("hello");}
let helloOnce = onlyOnceMaker(sayHello);
helloOnce();
helloOnce();
```

Outputs 'hello".

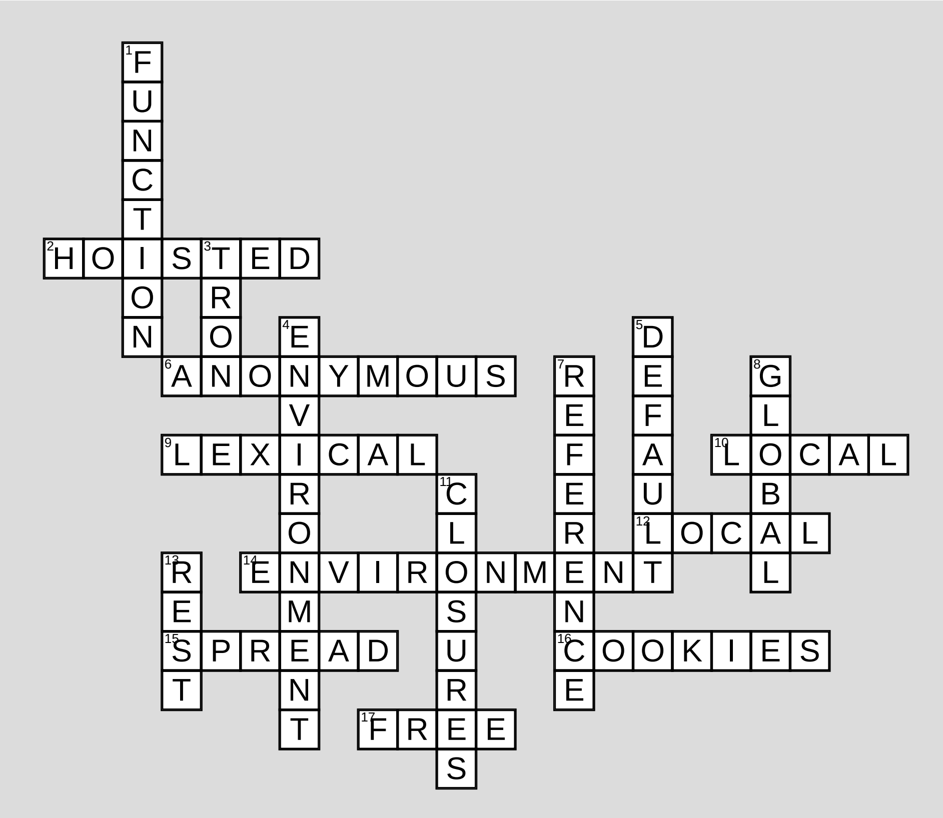
No output.

How would you test it?

JavaScript cross Solution

From page 534

Time for another crossword puzzle to burn some JavaScript into those neuron pathways. Here's our solution.


1 FUNCTION
2 HOISTED
3 TRON
4 ENVIRONMENT
5 DEFAULT
6 ANONYMOUS
7 REFERENCE
8 GLOBAL
9 LEXICAL
10 LOCAL
11 CLOSURES
12 LOCAL
13 REST
14 ENVIRONMENT
15 SPREAD
16 COOKIES
17 FREE

12 advanced object construction

Creating objects

HANDLE
WITH
CARE

So far, we've been crafting objects by hand. For each object, we've used an **object literal** to specify each and every property. That's okay on a small scale, but for serious code we need something better. That's where **classes** come in. With classes we can create objects much more easily, and we can create objects that all adhere to the same **design blueprint**—meaning we can use classes to ensure each object has the same properties and includes the same methods. And with classes we can write object code that is much more **concise** and a lot less error-prone when we're creating lots of objects. So, let's get started...

Creating objects with object literals

So far in this book, you've been using *object literals* to create objects. With an object literal, you create an object by writing it out…well, literally. Like this:

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,

    start() { this.started = true;},
    stop() { this.started = false;},
    drive() {
        // drive code here
    }
};
```

With an object literal, you type out each part of the object within curly braces. When you're done, the result is an actual JavaScript object, which you typically assign to a variable for later use.

Object literals give you a convenient way to create objects anywhere in your code, but if you need to create lots of objects—say, a whole fleet of taxis—you wouldn't want to have to type in a hundred different object literals, would you?

Brain Power

Think about creating a fleet of taxi objects. What other issues might using object literals cause?

☐ Tired fingers from a lot of typing!

☐ Can you ensure that each taxi has the same properties? What if you make a mistake or typo, or leave out a property?

☐ A lot of object literals means a lot of code. Isn't that going to lead to slow download times for the browser?

☐ The code for the start, stop, and drive methods would have to be duplicated over and over.

☐ What if you decide to add or delete a property (or to change the way start or stop works)? You'd have to make the change in all the taxi objects.

Using conventions for objects

The other thing we've been doing so far is creating objects *by convention*. For example, we've been putting properties and methods together and saying "it's a car" or "it's a dog," but the only thing that makes two such objects cars or dogs is that we've followed our own conventions.

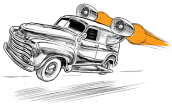
Again, this technique might work on a small scale, but it's problematic when we have lots of objects, or even lots of developers working on the same code who might not fully know or follow the conventions.

But don't take our word for it. Take a look at the following objects, which we've been told are all cars:

This taxi definitely looks like the cars we've been dealing with, but it has some extra properties and methods.

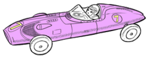
```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    fare: 2.50,

    start() { ...},

    stop() { ... },

    pay: function {
         // pay code here
    }
```

Okay, this looks a lot like our other car objects...but wait. This has racing features like rocket power!



```
let rocketCar = {
    make: "AlphaRocket",
    model: "xr-10a",
    year: 2098,
    color: "white",
    passengers: 6,
    convertible: false,
    mileage: 1938320,
    started: false,
    thrustOn: false,
    start() {
         this.started = true;
    },

    stop() {
         this.started = false;
    },

    drive() {
         // drive code here
    },

    thrust() {
         // code for thrust
    }
};
```

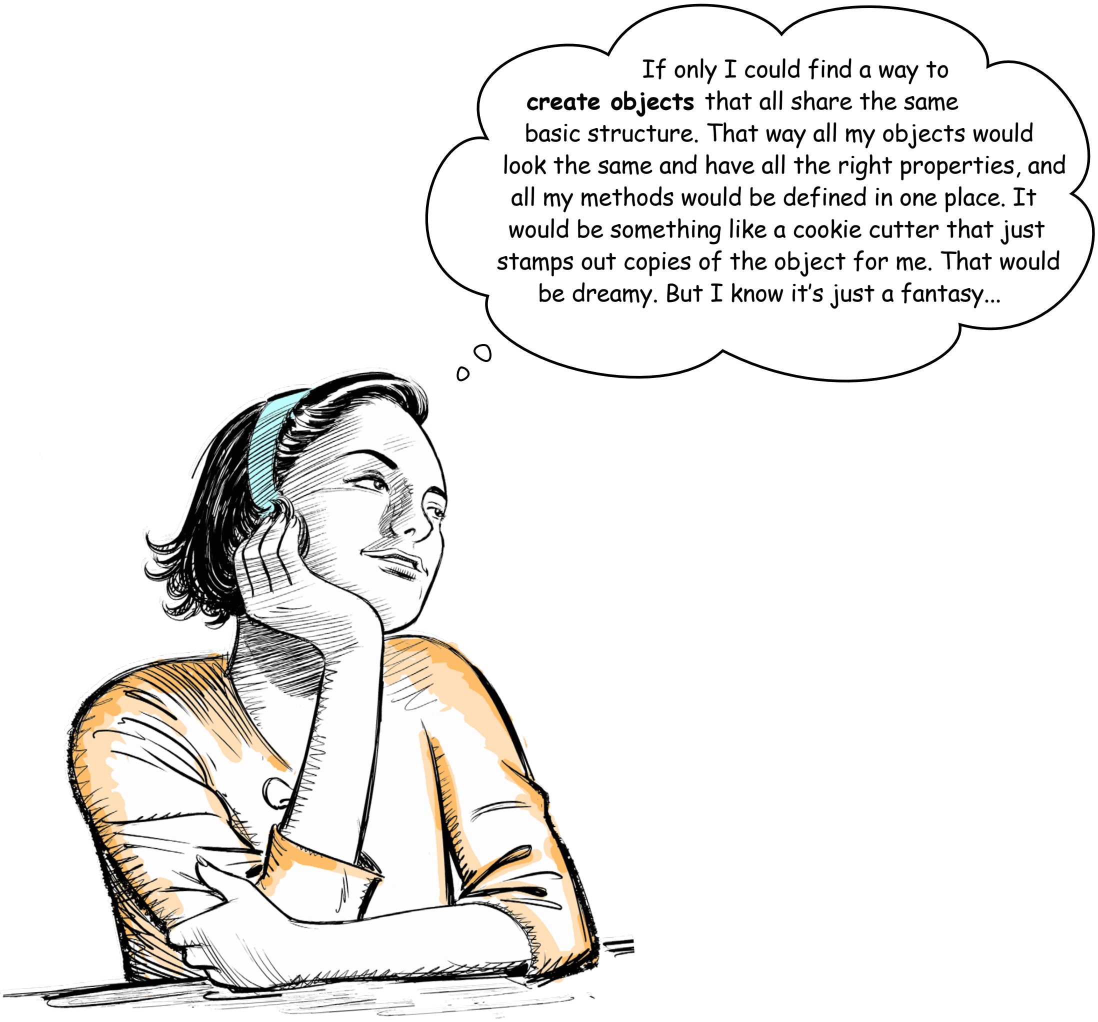
The chevy looks like a great car, but we're not seeing some of the basic properties it needs, like mileage or color. It also seems to have a few extra properties. That could be a problem...



```
let toyCar = {
    make: "Mattel",
    model: "PeeWee",
    color: "blue",
    type: "wind up",
    price: "2.99"
};
```

Wait, this might be a car, but it looks nothing like our other cars. It does have a make, model, and color, but it looks like a toy, not a car. What's this doing here?

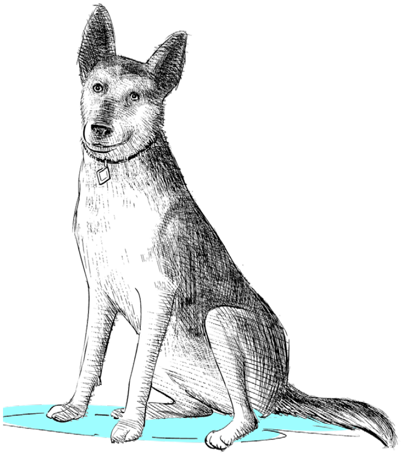
```
let chevy = {
    make: "Chevy",
    model: "Bel-Air",
    year: 1957,
    passengers: 2,
    convertible: false,
    started: false,
    oilLevel: 1.0,

    start() {
         if (oilLevel > .75) {
              this.started = true;
         }
    },

    stop() {
         this.started = false;
    },

    drive() {
         // drive code here
    }

};
```

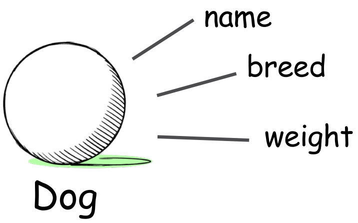

If only I could find a way to **create objects** that all share the same basic structure. That way all my objects would look the same and have all the right properties, and all my methods would be defined in one place. It would be something like a cookie cutter that just stamps out copies of the object for me. That would be dreamy. But I know it's just a fantasy...

Introducing classes

Classes are your path to better object creation. Think of a class like a little blueprint for creating an endless number of similar objects.

The best way to see how classes work is to create one. Let's revisit our old friend the dog object, and define a class to create as many dogs as we need. Here's a version of the dog object we've used before, with a name, a breed, and a weight.



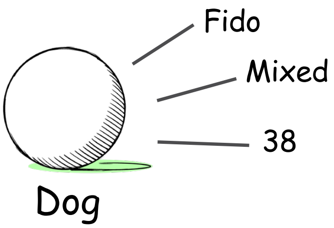


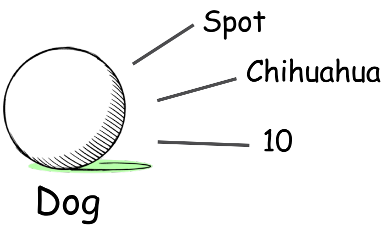
Now, if we were going to define such a dog with an object literal, it would look like this:

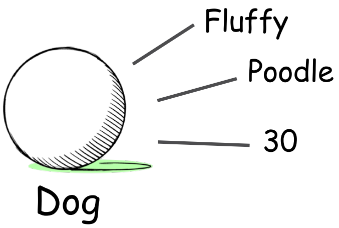
```
let dog = {
    name: "Fido",
    breed: "Mixed",
    weight: 38
};
```

Just a simple dog object created by an object literal. Now we need to figure out how to create a lot of these puppies.

But we don't want *just a Fido* dog, we want a way to create *any dog* that has a name, a breed, and a weight:

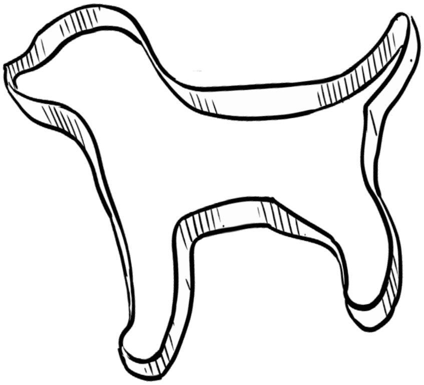






JavaScript classes are going to help us do that. Let's see how.

How to define a class



Creating an object from a class is a two-step process: first we define a class, and then we use it to create objects. Let's first focus on how to define a class.

What we want is a class that we can use to create dogs, and more specifically, dogs with names, breeds, and weights. So, we're going to define a class named Dog using the class keyword. We're also going to add a special function to the class, called the `constructor`, that helps us create dog objects.

Like this:

Notice that we start the class name with a capital letter. This isn't required; but everyone does it as a convention.

We use the **class** keyword to create the class.

Within the class we define a constructor function using the **constructor** keyword.

The parameters of the constructor function match the properties we want to supply for each individual dog.

```
class Dog {
    constructor(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
    }
}
```

The property names and parameter names don't have to be the same, but they often are—again, by convention.

Notice that this constructor function doesn't return anything.

Hmm, we're not using local variables like in most functions. Instead, we're using the **this** keyword, and we've only used that inside objects so far.

Hang on; we'll look at how we use the class next, and then all this is going to fall into place and make more sense.

Sharpen your pencil

We know we just introduced you to classes, but do a little pattern matching and see if you can write a class for creating Duck objects. You'll find one of our Duck object literals below to base your class on. Check your answer before moving on to make sure you get it.

```
let duck = {
    type: "redheaded",
    canFly: true
}
```

Here's an example duck object literal.

Write a class with a constructor function for creating ducks.

P.S. We know you haven't fully figured out how this all works yet, so for now concentrate on the syntax.

Solution on page 588

How to create an object from a class

Now that we have a Dog class, let's use it to create a new dog. To do that we use JavaScript's `new` operator, like this:

To create a Dog, we use the **new** operator with the class name...

...followed by the class name...

...and the arguments that get passed to the constructor function.

Try saying it out loud: "to create fido, I create a new Dog object with the name Fido that is a mixed breed and weighs 38 pounds."

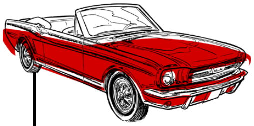
```
let fido = new Dog("Fido", "Mixed", 38);
```

So, to create a new dog object with a name of "Fido", a breed of "Mixed", and a weight of 38, we start with the `new` keyword and follow it with the class name, and then with the arguments we want to pass to the constructor function. After this statement is evaluated, the variable `fido` will hold a reference to our new dog object.

Now that we have a class for dogs, we can keep making them:

```
let fluffy = new Dog("Fluffy", "Poodle", 30);

let spot = new Dog("Spot", "Chihuahua", 10);
```

That's a bit easier than using object literals, isn't it? And by creating dog objects this way, we can ensure that each dog has the same set of properties: `name`, `breed`, and `weight`.



Test Drive

Let's get some quick hands-on experience to help this all sink in. Go ahead and put this code in a page and give it a test drive.

```
class Dog {
    constructor(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
    }
}

let fido = new Dog("Fido", "Mixed", 38);
let fluffy = new Dog("Fluffy", "Poodle", 30);
let spot = new Dog("Spot", "Chihuahua", 10);

let dogs = [fido, fluffy, spot];

dogs.forEach(dog => console.log(dog));
```

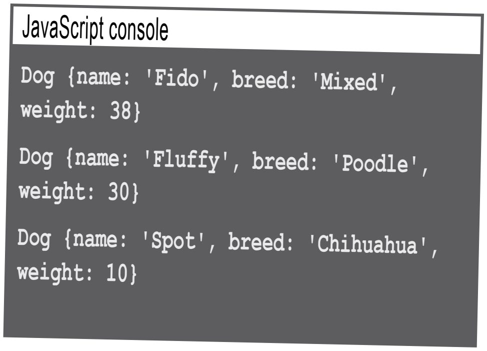
Here's our class definition.

Let's create a bunch of new dogs.

Next, we'll throw them in an array.

And then we interate over the dogs array and print each dog.

Now we have three different Dog objects, all created from the same blueprint, our Dog class.



```
Dog {name: 'Fido', breed: 'Mixed',
weight: 38}
Dog {name: 'Fluffy', breed: 'Poodle',
weight: 30}
Dog {name: 'Spot', breed: 'Chihuahua',
weight: 10}
```

How classes work

PRODUCTION: Head First
SCENE 2
TAKE 3
ROLL
SOUND
DATE
PRODUCTION CO. O'Reilly
DIRECTOR
Behind the Scenes

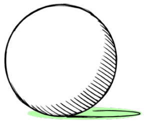
We've seen how to declare a class and how to use it to create objects, but we should also take a look behind the scenes to see how a class actually works. Here's the key: to understand classes we need to know what the `new` operator is doing.

We'll begin with the statement we used to create `fido`:

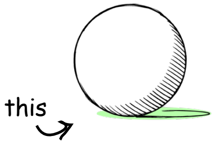
```
let fido = new Dog("Fido", "Mixed", 38);
```

Take a look at the righthand side of the assignment, where all the action is. Let's follow its execution:

① The first thing `new` does is create a new, empty object.



② Next, `new` sets `this` to point to the new object.

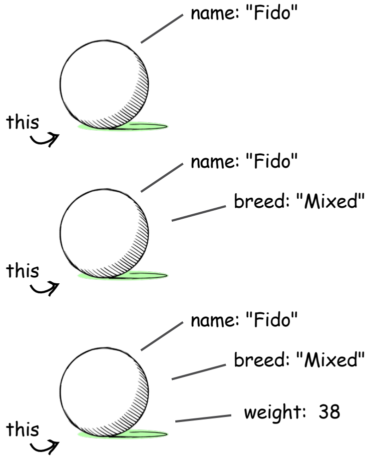


Remember from Chapter 5 that **this** holds a reference to the current object our code is dealing with.

③ With `this` set up, the constructor is called and passed "Fido", "Mixed", and 38 as arguments.

```
            "Fido"     "Mixed"     38

class Dog {
  constructor(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
  }
}
```

④ Next, the body of the function is invoked. Like most constructors, Dog assigns values to properties in the newly created object by using `this`.

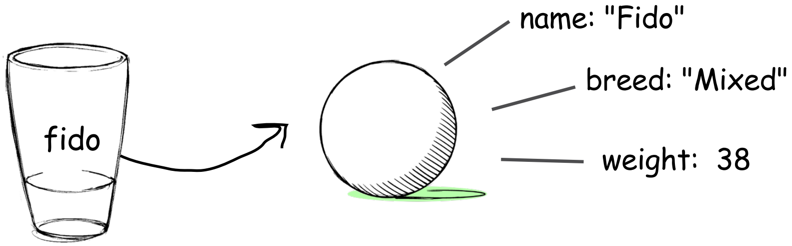

name: "Fido"
this
name: "Fido"
breed: "Mixed"
this
name: "Fido"
breed: "Mixed"
weight: 38
this

Executing the body of the Dog function customizes the new object with three properties, assigning them the values of the respective parameters.

PRODUCTION: Head First
SCENE 2
TAKE 3
ROLL
SOUND
DATE
PRODUCTION CO. O'Reilly
DIRECTOR

Behind the Scenes

⑤ Finally, once the Dog class constructor function has completed, the `new` operator returns `this`, which is a reference to the newly created object. Notice `this` is returned for you; you don't have to explicitly return it in your code. And after the new object has been returned, we assign it to the variable `fido`.


fido
name: "Fido"
breed: "Mixed"
weight: 38

BE the Browser

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors. Look at the solution at the end of the chapter to see if you found them all. Feel free to make style comments too. You've earned the right.

```
class widget {
    constructor(partNo, size) {
        let this.no = partNo;
        let this.breed = size;
    }
}
class FormFactor {
    constructor(material, widget) {
        this.material = material;
        this.widget = widget;
        return this;
    }
}
let widgetA = widget(100, "large");
let widgetB = new widget(101, "small");
let formFactorA = newFormFactor("plastic", widgetA);
let formFactorB = new ForumFactor("metal", widgetB);
```

→ Solution on page 588

there are no Dumb Questions

Q: Why do class names start with a capital letter?

A: This is a convention that JavaScript developers use so they can easily identify classes from their names in code (say, compared to variable names and function names).

Q: Other than setting up the properties of the `this` object, is a class's constructor function just like a regular object method?

A: If you mean computationally, yes. You can do anything in a constructor you can do in a regular function, like declare and use variables, use for loops, call other functions, and so on. The only thing you don't want to do is return a value (other than **this**) from a class's constructor, because that will cause the constructor to not return the object it's supposed to be constructing.

Q: Do the parameter names of a class's constructor function have to match the property names?

A: No. You can use whatever names you want for the parameters. The parameters are just used to hold values that we want to assign to the object's properties to customize the object. What matters are the names of the properties you use for the object. That said, we often do use the same names for clarity, so we know which properties we're assigning by looking at the constructor function definition in a class.

Q: Is it better to create objects with a class than with object literals?

A: Both are useful. A class is useful when you want to create lots of objects with the same property names and methods. Using them is convenient, promotes code reuse, and provides consistency across your objects.

But sometimes we just need a quick object, perhaps a one-time-use-only object, and literals are a good option here; they are concise and expressive.

So it really depends what your needs are. Both are great ways to create an object.

Q: I'm still a bit confused by `this` in a class's constructor function. We're using `this` to assign properties to the object, and we're also using `this` in the methods of the object. Are these the same thing?

A: When you use a class to create an object the value of **this** is set to the new object, so all the code that is evaluated in the constructor applies to that new object.

Later, when you call a method on an object, **this** is set to the object whose method you called. So, the **this** in your methods will always refer to the object whose method was called.

Q: Is an object created by a class just like an object created with a literal?

A: Mostly. We'll get into some differences shortly.

A class is useful when you want to create lots of objects with the same property names and methods. Using classes is convenient, promotes code reuse, and provides consistency across your objects.

Let's add some methods

The dog objects that the Dog class creates are just like the dogs from earlier in the book…except that our newly constructed dogs can't bark (because they don't have a `bark` method yet). This is easily fixed, because in addition to assigning values to properties in the constructor function, we can set up methods in the class too. Let's update the code to include a `bark` method:

```
class Dog {
    constructor(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
    }
    bark() {
        if (this.weight > 25) {
            console.log(`${this.name} says Woof!`);
        } else {
            console.log(`${this.name} says Yip!`);
        }
    }
}
```

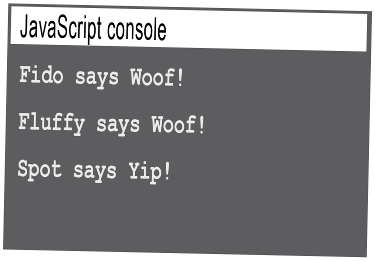
To add a bark method, we add the function to the class using its name and a code block. Note: the **function** keyword is not needed in this method definition.

Now every dog object you make with the Dog class will also have a bark method that you can invoke.

Test Drive

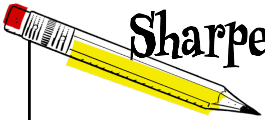
Let's add the code above to an HTML page, and then add the code below to test it:

```
let fido = new Dog("Fido", "Mixed", 38);
let fluffy = new Dog("Fluffy", "Poodle", 30);
let spot = new Dog("Spot", "Chihuahua", 10);
let dogs = [fido, fluffy, spot];

dogs.forEach(dog => dog.bark());
```



```
Fido says Woof!
Fluffy says Woof!
Spot says Yip!
```

Make sure your dog objects bark like they're supposed to.



Sharpen your pencil

Let's make sure you've got the subtleties of object literals and classes down. You'll find a dog object literal on the top, and a Dog class on the bottom. Compare and contrast. In what ways are they alike and different? Be sure and check your answers before moving on.

```
let fido = {
    name: "Fido",
    breed: "Mixed",
    weight: 38,
    bark() {
        if (this.weight > 25) {
            console.log(`${this.name} says Woof!`);
        } else {
            console.log(`${this.name} says Yip!`);
        }
    }
};
```

```
class Dog {
    constructor(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
    }
    bark() {
        if (this.weight > 25) {
            console.log(`${this.name} says Woof!`);
        } else {
            console.log(`${this.name} says Yip!`);
        }
    }
}

let fido = new Dog("Fido", "Mixed", 38);
```

→ Solution on page 589

The Class Exposed

This week's interview: Getting to know class

Head First: `class`, where have you been hiding? How did we get to Chapter 12 before seeing you?

class: There are still a lot of scripts out there that don't use me, or use me without understanding me.

Head First: Why is that?

class: Because many scripters just use object literals or copy and paste code that uses me, without understanding how I work.

Head First: That's a good point…object literals are convenient, and I myself am not quite clear on when or how to use you just yet.

class: Well it's true, I am kind of an advanced feature. After all, to know how to use me, you first have to know how objects work, and how functions work, and how `this` works...it's a lot to wrap your head around before you even learn about me at all!

Head First: Can you give us the elevator pitch about yourself? Now that our readers know about objects, functions, and `this`, it would be great for them to get motivated for learning about you.

class: Let me think for a second…Okay, here you go: I work with the `new` operator to create new objects.

Head First: Umm, I hate to break it to you, but that isn't the best elevator pitch.

class: Gimme a break...I'm a language feature; I'm not in PR.

Head First: Well, you do raise several questions with that pitch. First of all, why do you need an operator?

class: It's just the way I work. `new` and I work together. `new` says "create a new, empty object, and then call my constructor function." Then my constructor function does all the work of setting up the properties for the new object. We make a good team.

Head First: Wait, that was too fast. One step at a time, please.

class: Well, first, `new` makes a new object. Everyone thinks that my constructor function is what does it, but it's actually `new`. I often say `new` is seriously underrated.

Head First: Go on…

class: Okay, so `new` then calls my constructor function and makes sure that the new object is referenced by the `this` keyword in the body of my function.

Head First: Why?

class: So that the statements in the body of my constructor function have a way to refer to the object. After all, the whole point of a constructor function is to extend that object using new properties and methods. If you're using the constructor to create objects like dogs and cars, you're going to want those objects to have some properties, right?

Head First: Right. And then?

class: Then I make sure that the new object that was created is returned from the constructor. It's a nice convenience so that developers don't have to remember to return it themselves.

Head First: It sounds very convenient. Now why would anyone use an object literal after learning about you?

class: Oh, object literal and I go way back. He's a great guy, and I'd use him in a second if I had to create a quick object. But you want me when you've got to create a lot of similar objects, when you want to make sure your objects are taking advantage of code reuse, when you want to ensure some consistency, and, after you've learned a little more, to support some even more advanced uses.

Head First: More advanced? Oh, do tell!

class: Now now, let's keep these readers focused. They've barely even started learning about me!

Exercise

We've got a class to create coffee drinks, but it's missing its methods.

We need a method, getSize, that returns a string depending on the number of ounces of coffee:

- 8 oz is a small
- 12 oz is a medium
- 16 oz is a large

We also need a method, toString, that returns a string that represents your order, like "You've ordered a small House Blend coffee."

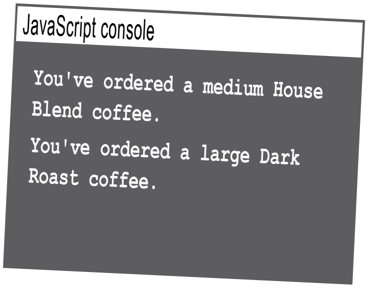
Write your code below, and then test it in the browser. Try creating a few different sizes of coffee. Check your answer before you go on.

```
class Coffee {
    constructor(roast, ounces) {
        this.roast = roast;
        this.ounces = ounces;
    }
```

Write the two methods for this constructor here.

```
}

let coffee = new Coffee("House Blend", 12);
console.log(coffee.toString());

let darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

Here's our output; yours should look similar.

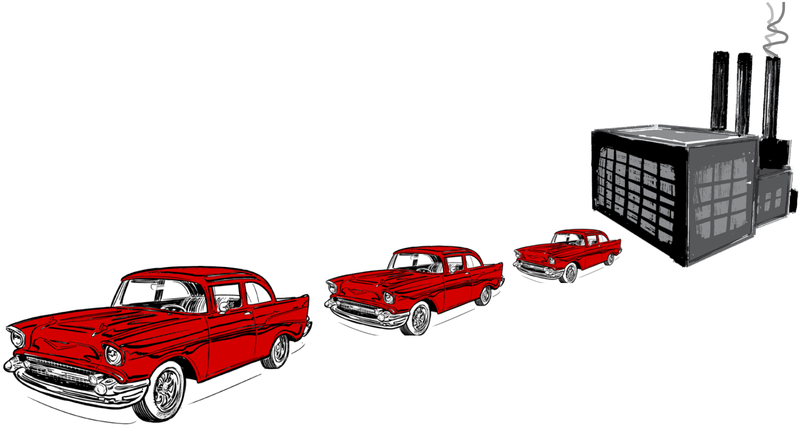


JavaScript console

```
You've ordered a medium House
Blend coffee.
You've ordered a large Dark
Roast coffee.
```

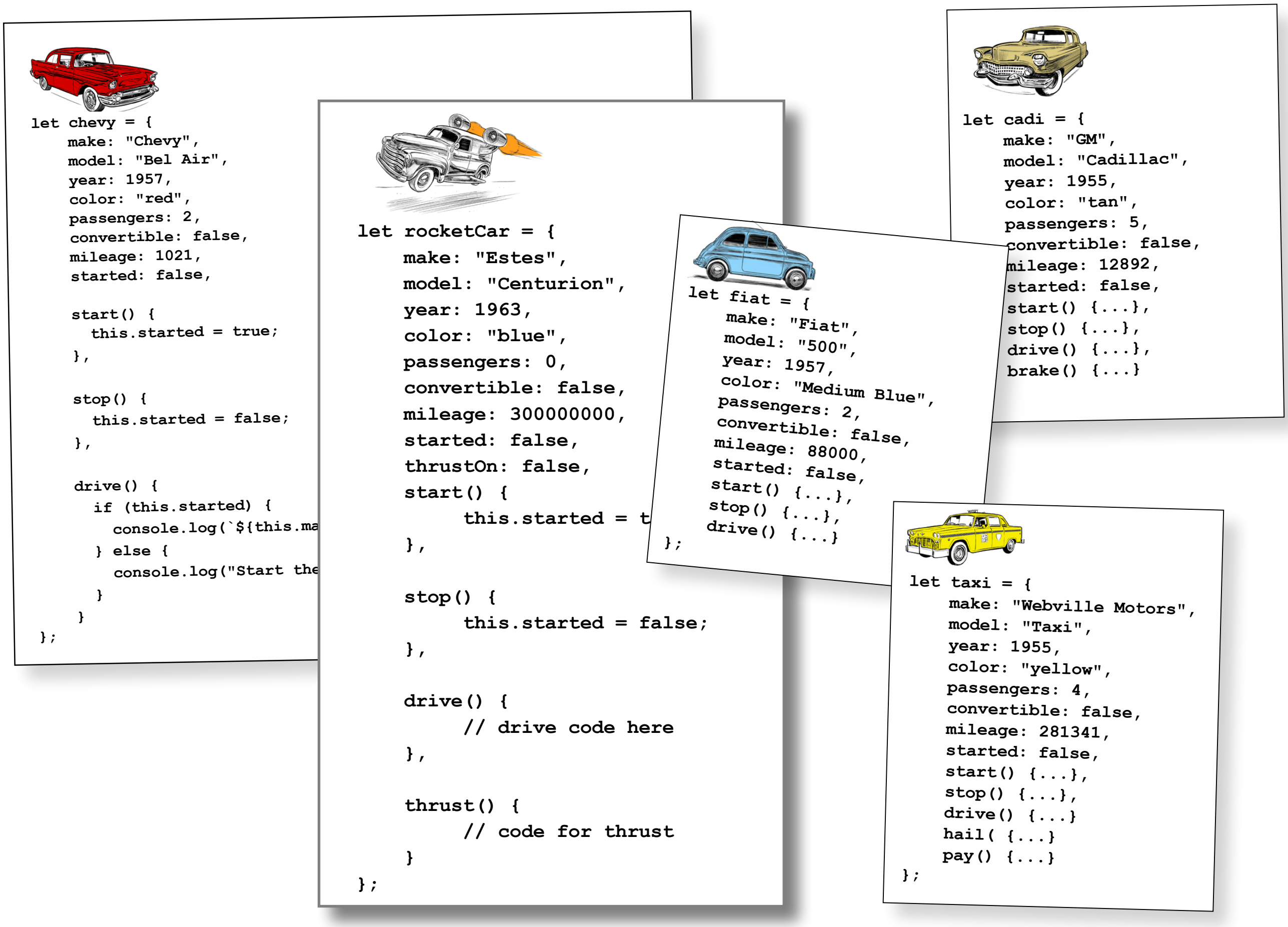
Solution on page 590

It's production time!



You've learned your object construction skills just in time, because we've got a project for you: building a car simulator. We're going to take all the car literals we've used so far in the book and build a nice little simulator environment for them to run in. Chevys, taxis, rocket cars? It doesn't matter, our classes are going to let us create all of those and more.

Check out the various kinds of cars we are going to need support below. Notice we've already taken the liberty of making most of their properties and methods uniform, but they still have some differences. Go ahead and take a look, and then let's build a class that can create car objects for any kind of car that has these property names and methods.



```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,

    start() {
      this.started = true;
    },

    stop() {
      this.started = false;
    },

    drive() {
      if (this.started) {
        console.log(`${this.ma
      } else {
        console.log("Start the
      }
    }
};
```

```
let rocketCar = {
    make: "Estes",
    model: "Centurion",
    year: 1963,
    color: "blue",
    passengers: 0,
    convertible: false,
    mileage: 300000000,
    started: false,
    thrustOn: false,
    start() {
          this.started = t
    },

    stop() {
          this.started = false;
    },

    drive() {
          // drive code here
    },

    thrust() {
          // code for thrust
    }
};
```

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,
    start() {...},
    stop() {...},
    drive() {...}
};
```

```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start() {...},
    stop() {...},
    drive() {...},
    brake() {...}
```

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start() {...},
    stop() {...},
    drive() {...}
    hail( {...}
    pay() {...}
};
```

I kinda get how we're going to build a simulator, but some of these cars are quite different; how do we manage that with classes?

Sam: I noticed that too. The taxi and rocket car really stood out. You can hail a taxi and also pay for it. And the rocket car has thrust. How do we create a class for a normal car, and not have it have those behaviors?

Mara: I think we're going to have to explore *inheritance*.

Alex: Inhera-what?

Mara: Object-oriented inheritance; it's a way to have objects inherit the properties and methods of another class while extending what it does.

Sam: How does that help?

Mara: Think about it like this: we can define a basic Car class that can be used by the fiat, the chevy, and the cadi. Then we can have our taxi inherit all the basic Car properties and methods, and extend them with hail and pay methods along with new properties it might need, like occupied and fare amount.

Alex: That sounds promising. How do we do it?

Mara: Let's sit down and see if we can work through it.

Sam, Alex: We're in!

The basic Car class

Let's begin by creating a basic Car class. Looking at our design two pages back, the cars all need make, model, year, color, passengers, convertible, and mileage properties. They also need stop, start, drive, and brake methods. We'll worry about the taxi and rocket car's special features in a bit.

Here's our basic Car class:

```
class Car {
    constructor(make, model, year, color, passengers, convertible, mileage) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
        this.passengers = passengers;
        this.convertible = convertible;
        this.mileage = mileage;
        this.started = false;
    }
    start() {
        this.started = true;
        console.log(`The ${this.make} ${this.model} has started`);
    }
    stop() {
        this.started = false;
        console.log(`The ${this.make} ${this.model} has stopped`);
    }
    drive() {
        if (this.started) {
            console.log(`The ${this.make} ${this.model} goes zoom zoom!`);
        } else {
            console.log(`You need to start the ${this.make} ${this.model} first`);
        }
    }
    brake() {
        console.log(`The ${this.make} ${this.model} is braking`);
    }
}
```

Here are all our properties. Wow, there are a lot of them, and a lot of parameters; maybe we can do something to clean that up in a bit.

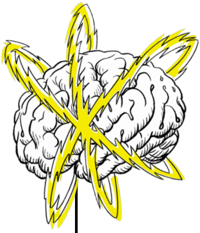
And check out our methods, all ready for the simulator.

Starting...

Stopping the car...

Driving. Notice you can't drive without starting the car first.

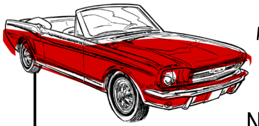
And the all-important braking.



Brain Power

Can you see how we're using properties and methods together to make objects that feel like real cars? For instance, you can't drive a car until you start it. What other enhancements would you add to make this more car-like?

Your ideas here.



Test Drive

Now that we have a basic car implemented, let's write a little simulator code for it:

```
let chevy = new Car("Chevy", "Bel Air", 1957, "red", 2, false, 1021);
let cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
let fiat = new Car("Fiat", "500", 1957, "Medium Blue", 2, false, 88000);

let cars = [chevy, cadi, fiat];

cars.forEach(car => {
    car.drive();
    car.start();
    car.drive();
    car.brake();
    car.drive();
    car.stop();
});
```

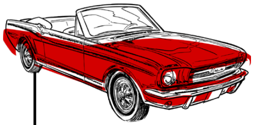
Are you starting to see how easy creating new objects can be with constructors?

Let's create one each of our basic cars; we'll save the taxi and rocket car for later.

Then we'll throw them all in an array so we can iterate over them for quick testing.

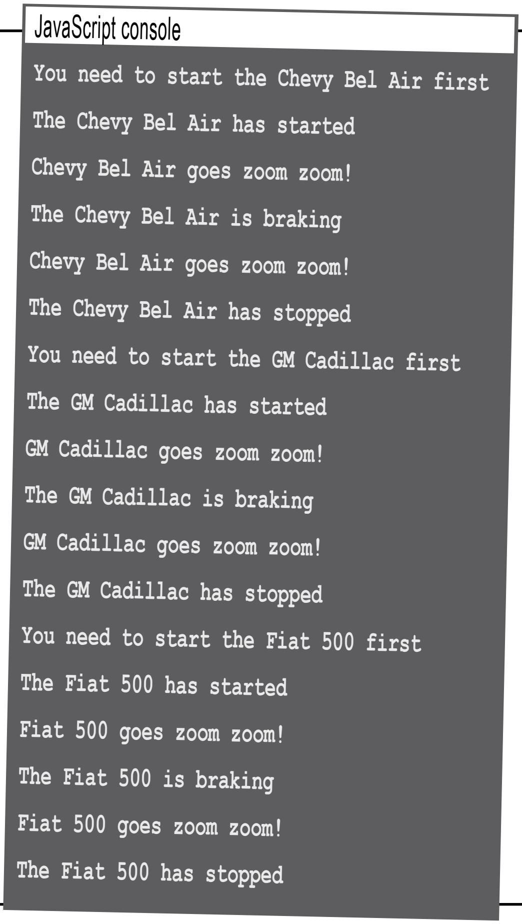
And then let's exercise the various methods, including trying to drive the car without starting it.

Get all this code entered. You'll need to start by putting the Car constructor in an HTML page, then add the test code and give it a run. What did you get? Our output is on the next page.



Test Drive Solution

Here's our output; it looks like our simulator is off to a good start.



```
JavaScript console
You need to start the Chevy Bel Air first
The Chevy Bel Air has started
Chevy Bel Air goes zoom zoom!
The Chevy Bel Air is braking
Chevy Bel Air goes zoom zoom!
The Chevy Bel Air has stopped
You need to start the GM Cadillac first
The GM Cadillac has started
GM Cadillac goes zoom zoom!
The GM Cadillac is braking
GM Cadillac goes zoom zoom!
The GM Cadillac has stopped
You need to start the Fiat 500 first
The Fiat 500 has started
Fiat 500 goes zoom zoom!
The Fiat 500 is braking
Fiat 500 goes zoom zoom!
The Fiat 500 has stopped
```

Did you add your own car to the mix? Try changing what the cars do (like driving before the car is started). Or maybe you can make the number of times we call the drive method random?

Sam: Okay, now it's time for that taxi. This is where I guess the inheritance is going to come in?

Mara: Yup. So a taxi is pretty much a car, right? The only difference is it picks up passengers, collects payment, and is usually bright yellow.

Alex: I'm not so worried about the yellow part.

Mara: Just kidding. So, what we can do is *extend* the Car class and create a taxi that does everything like a car, except we'll add in the ability to call it for a ride and to pay it for the ride.

Sam: That doesn't sound so bad; let's give it a try.

Implementing the Taxi class with extends

We want a taxi to be almost exactly like our basic car, except that the taxi will have two extra properties: `occupied` and `fare`. The `occupied` property will be true if the taxi is occupied and false otherwise; the `fare` property will hold the amount a taxi charges for a ride. We'll also add a couple of extra methods.

We can reuse the Car class and extend it with new properties and methods for a Taxi class using the `extends` keyword, like this:

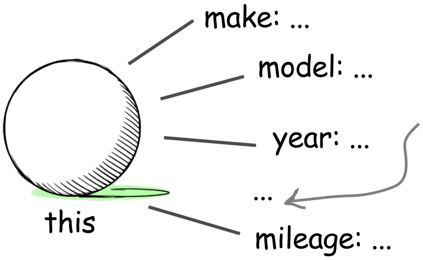
We create a Taxi class in the usual way...

...except we tell the Taxi class to extend the Car class.

The constructor has all the same parameters as the Car constructor, plus one extra, fare.

```
class Taxi extends Car {
    constructor(make, model, year, color, passengers, convertible, mileage, fare) {

        super(make, model, year, color, passengers, convertible, mileage);
```

The first thing we do in the constructor is call a special function, super. This calls the constructor method of the class we're extending—in this case, Car. We pass in all the values we need to construct a Car, and super makes a new object that is now "this" in Taxi: that is, a new object that has a make, model, year, and all the other properties of a Car.



color, passengers, etc. of a Car are here but not shown.

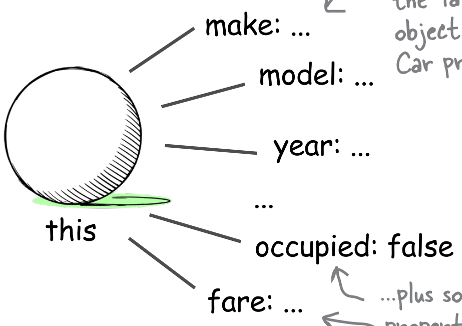
Note: we haven't added fare or occupied yet.

Now that we've got an object with the properties of a Car, we need to add on the properties that are unique to a Taxi: occupied and fare.

```
        this.occupied = false;
        this.fare = fare;
    }
}
```

We're assigning an intial value of false to occupied and setting fare to the value that was passed into the constructor function.

What we have at the end of the constructor function (and what will be returned when we call **`new Taxi(...)`**) is a Taxi object that has all the properties of a Car, plus a couple of extras. We got to reuse the Car constructor by calling `super(...)`, and we also get to reuse all the methods from Car and add new methods unique to taxis. Let's see how to do that, next.



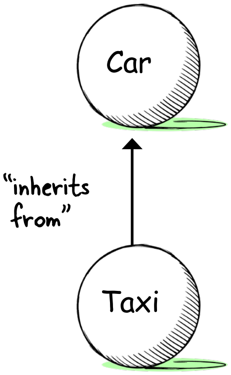
What we get with the Taxi class is an object with all the Car properties...

...plus some Taxi properties too.

Adding new methods to the Taxi class

Okay, we've got a good start to our Taxi class, but we still need to add a couple of methods that are unique to Taxi objects: a `hail` method to call a taxi, and a `pay` method to pay the taxi.

Remember that Taxi objects will *inherit* all the Car properties and methods from Car, so we only need to add the methods unique to taxis in the Taxi class:



Taxi inherits from Car, including both properties and methods.

And Taxi extends Car by adding new properties and methods.

Here's our Taxi class from the previous page, only now we're adding two new methods: hail and pay.

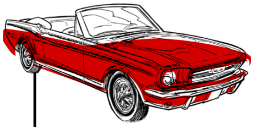
```
class Taxi extends Car {
    constructor(make, model, year, color, passengers, convertible, mileage, fare) {
        super(make, model, year, color, passengers, convertible, mileage);
        this.occupied = false;
        this.fare = fare;
    }

    hail() {
        if (!this.occupied) {
            console.log(`The ${this.make} ${this.model} is here`);
            this.occupied = true;
        } else {
            console.log(`Sorry this ${this.make} ${this.model} is busy`);
        }
    }

    pay() {
        if (this.occupied) {
            console.log(`The ${this.make} ${this.model} says that will be
                        \$${this.fare.toFixed(2)}`);
            this.occupied = false;
        } else {
            console.log(`The ${this.make} ${this.model} has no customers`);
        }
    }
}
```

The hail method checks to see if the taxi is occupied; if it's not, the taxi arrives, and we set occupied to true.

Otherwise, we tell the rider the taxi's busy.

The pay method checks to see if the taxi is occupied (by the rider), and if so asks for the amount of the fare and sets occupied to false (so the taxi is ready for another rider).

If we call the pay method when the taxi is not occupied, we can't pay the fare!



Test Drive

Let's test our Taxi class! We create a new Taxi object in the usual way, passing all the arguments we'd pass to Car, plus one extra at the end: the fare, in this case $2.50. Then we put the Taxi through its paces. We call some methods Taxi inherits from Car (like start and drive) and the methods unique to Taxi (hail and pay).

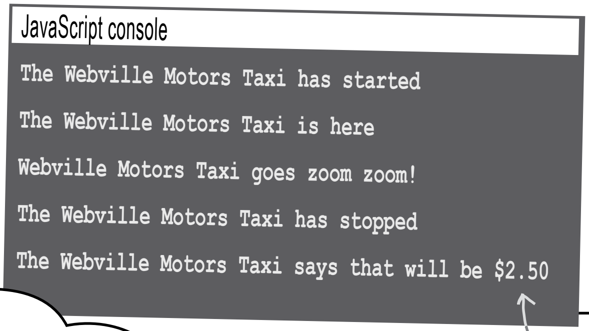
```
let taxi = new Taxi("Webville Motors", "Taxi", 1961, "yellow", 3, false, 7340, 2.50);
taxi.start();
taxi.hail();
taxi.drive();
taxi.stop();
taxi.pay();
```

Don't forget the extra argument for the fare.

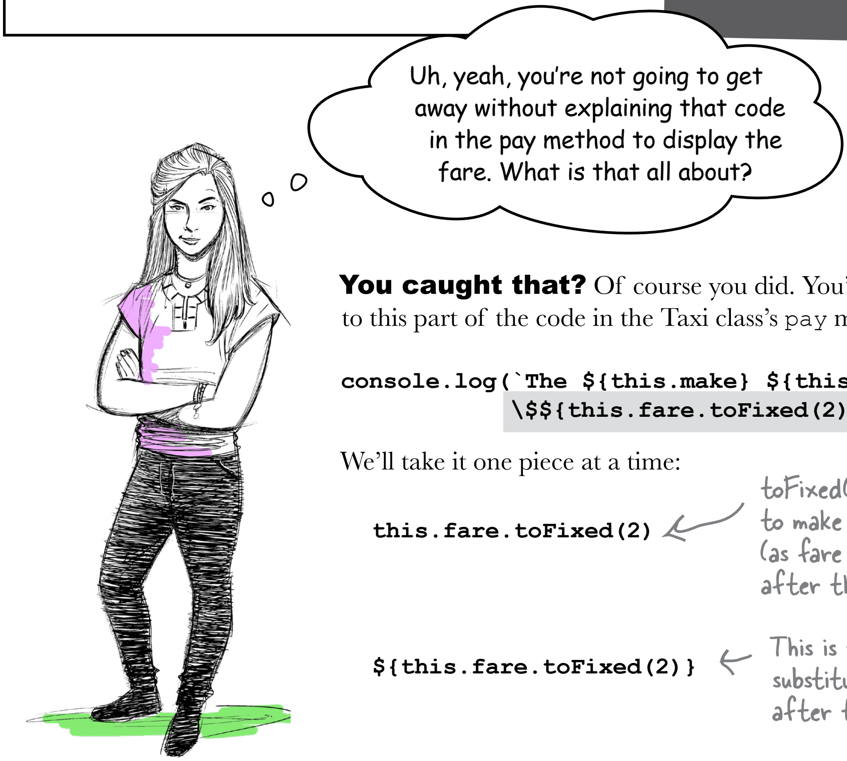
These methods the Taxi class inherits from Car...

...and these methods are unique to Taxi.

Our results.



```
The Webville Motors Taxi has started
The Webville Motors Taxi is here
Webville Motors Taxi goes zoom zoom!
The Webville Motors Taxi has stopped
The Webville Motors Taxi says that will be $2.50
```



Uh, yeah, you're not going to get away without explaining that code in the pay method to display the fare. What is that all about?

You caught that? Of course you did. You're probably referring to this part of the code in the Taxi class's `pay` method:

```
console.log(`The ${this.make} ${this.model} says that will be
             \$${this.fare.toFixed(2)}`);
```

We'll take it one piece at a time:

`this.fare.toFixed(2)`

toFixed(2) is a method you can call on numbers to make sure when they're converted to strings (as fare is here) they display with 2 digits after the decimal point.

`${this.fare.toFixed(2)}`

This is the usual template literal that says substitute the value of this.fare (with 2 digits after the decimal!) into the string.

`\$${this.fare.toFixed(2)}`

And \$ says print a $ in front of the fare. We can't write just $ because that has a special meaning in a template literal; so we have to use \$ to get $2.50.

The Taxi is looking great! Can we tackle the RocketCar next? We'll need a constructor to start off with...

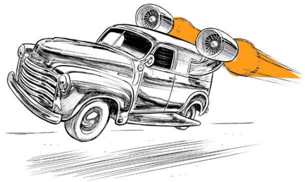
Exercise

Why don't you take a crack at the code for the RocketCar constructor? Like Taxi, RocketCar will extend the Car class. It needs just one extra property, thrustOn, which will be initialized to false. No peeking; our solution is on the next page.

Your code here for the RocketCar class with constructor (we'll add the new methods shortly).

Solution on page 591

Implementing the RocketCar class



The RocketCar class will start off looking a lot like Taxi. We'll extend the Car class, create a constructor function with all the parameters we need (but no extra `fare` parameter is needed here), and add one new property, `thrustOn`, in the constructor, like this:

```
class RocketCar extends Car {
    constructor(make, model, year, color, passengers, convertible, mileage) {
        super(make, model, year, color, passengers, convertible, mileage);
        this.thrustOn = false;
    }
}
```

The basic constructor for RocketCar.

We add one new property, thrustOn, in the constructor and initialize it to false.

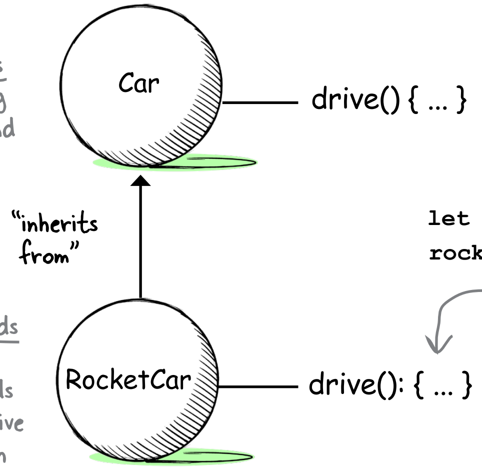
Hopefully this code matches yours from the exercise on the previous page.

Overriding Car's drive method

Next, let's add the methods. We want the RocketCar to drive sort of like a Car, except the RocketCar has thrusters, and if the thrusters are on, then the RocketCar will drive much faster than a regular Car. So, we need to give the RocketCar its own `drive` method.

We know that the RocketCar class inherits the `drive` method from Car, but that drive method isn't quite right. We'll implement our own `drive` method in RocketCar that *overrides* the Car class's `drive` method, so if we call the `drive` method on a RocketCar object we'll get the behavior defined in RocketCar's `drive` method, not the behavior defined in Car's `drive` method:

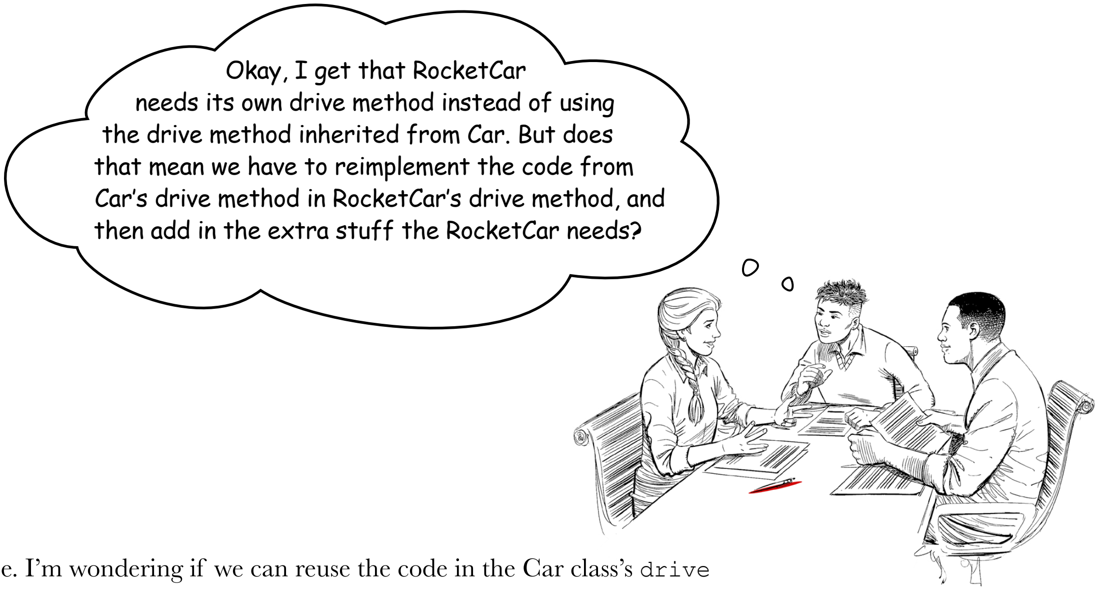
RocketCar inherits from Car, including both properties and methods.



```
let rocketCar = new RocketCar(...);
rocketCar.drive();
```

And RocketCar extends Car by adding new properties and methods AND overrides the drive method from Car with its own drive method.

When we call RocketCar's drive method, we'll get the behavior defined in RocketCar, not the behavior defined in Car.


Okay, I get that RocketCar needs its own drive method instead of using the drive method inherited from Car. But does that mean we have to reimplement the code from Car's drive method in RocketCar's drive method, and then add in the extra stuff the RocketCar needs?

Mara: Maybe. I'm wondering if we can reuse the code in the Car class's `drive` method using `super`.

Alex: You mean like we do in the constructor? I thought that was a special constructor thing.

Mara: No, I think we can use it whenever we want to refer to a method in a class's superclass.

Sam: Superclass?

Mara: Right, the class that another class is extending with the `extends` keyword is called a superclass. That's why we refer to it using `super`.

Alex: Oh...interesting.

Mara: We should be able to write:

```
super.drive();
```

to refer to Car's drive method.

Sam: But we didn't write:

```
super.constructor();
```

when we wanted to call Car's constructor, right? We just wrote:

```
super(...);
```

Mara: Right. The constructor is a special case. If you say `super()` you're referring to the constructor, but if you say `super.drive()` you're referring to a specific method.

Sam: Why can't we just write `this.drive()`?

Mara: Because that would refer to the `drive` method in the RocketCar class, and we want to refer to the `drive` method in the Car class.

Sam: Ohhh...

Mara: Let's give `super.drive()` a try and see if it works.

Finishing up the RocketCar class

Okay, it's time to finish the RocketCar class. We have the constructor in place, and we have a general idea of what we want the `drive` method to do—behave like Car's `drive` method but with extra thrust—so let's write the code. We're going to add the `drive` method along with a `thrust` method to turn on and off the thruster:

Here's the constructor we already wrote.

```
class RocketCar extends Car {
    constructor(make, model, year, color, passengers, convertible, mileage) {
        super(make, model, year, color, passengers, convertible, mileage);
        this.thrustOn = false;
    }
```

And here's the drive method. We want the RocketCar to drive like a Car, but add on extra speed if the thruster is on.

```
    drive() {
        super.drive();
        if (this.thrustOn) {
            console.log(`And the ${this.make} ${this.model} rocket thruster is on`);
        }
    }
```

So, first we call super.drive(). This calls Car's drive method, so we should see the RocketCar zoom-zooming just like a regular Car.

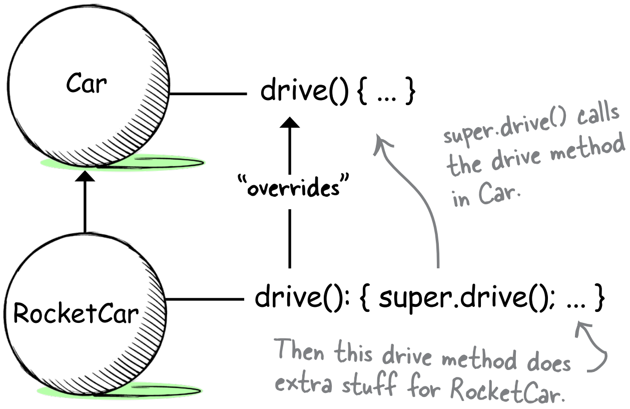
And here, we add on extra speed if the thruster is on.

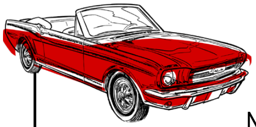
```
    thrust() {
        if (!this.thrustOn) {
            this.thrustOn = true;
        } else {
            this.thrustOn = false;
        }
    }
}
```

The thrust method is used to turn the thruster on and off. So, if we want the RocketCar to go extra fast, we'll turn on the thruster before we drive the RocketCar.

What's happening here?

Like Taxi, RocketCar extends Car, so it *inherits* the Car class's properties and methods. RocketCar *extends* Car by adding new properties and methods (`thrustOn` and `thrust`), and *overrides* the `drive` method from Car so it can drive differently than a Car. But we reuse Car's `drive` code by calling `super.drive()` in RocketCar's `drive` method, before adding special `drive` behavior for the RocketCar.


Car
drive() { ... }
"overrides"
super.drive() calls the drive method in Car.
RocketCar
drive(): { super.drive(); ... }
Then this drive method does extra stuff for RocketCar.



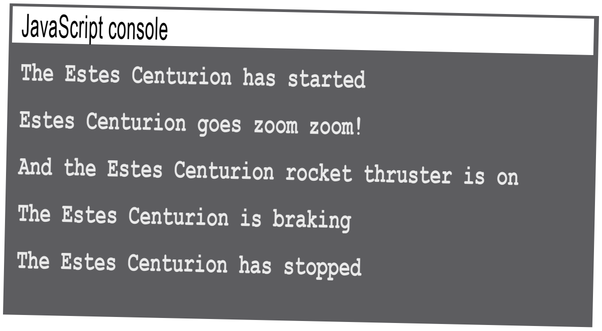
Test Drive

Now that we have the RocketCar class implemented, let's write a little test code for it:

```
let rocketCar = new RocketCar("Estes", "Centurion", 1963, "blue",
                                           0, false, 300000000);
rocketCar.start();
rocketCar.thrust();
rocketCar.drive();
rocketCar.brake();
rocketCar.stop();
```

Here's our rocketCar object made using the RocketCar class.

We put the rocketCar through its paces. We turn thrust on before we drive the rocketCar, so when we drive it, we see it's going extra fast with the thruster.



```
The Estes Centurion has started
Estes Centurion goes zoom zoom!
And the Estes Centurion rocket thruster is on
The Estes Centurion is braking
The Estes Centurion has stopped
```

there are no Dumb Questions

Q: Can you explain the difference between extends and inherits again? I'm not quite sure I've got it right.

A: We use the extends keyword to specify that a class inherits its properties and methods from another class, and potentially adds new properties and methods. For instance, the Taxi class extends the Car class, and, more informally, it inherits its properties and methods from the Car class. The Taxi class also adds new properties and methods to those it inherited from the Car class.

Q: So the class that's being extended is the superclass, right? What's the class that's doing the extending called?

A: We call the class that extends a superclass the subclass. So Car is the superclass of Taxi and RocketCar, and Taxi and RocketCar are subclasses of Car.

Q: So if you define a method in a superclass and define a method with the same name in the subclass, the method in the subclass always overrides the method in the superclass?

A: That's right. So, if you create a Car object, like a fiat, using the Car class, if you call fiat.drive() you'll get the behavior of Car's drive method. If you create a RocketCar object, because RocketCar extends Car and overrides drive, if you call rocketCar.drive() you'll get the behavior defined in the RocketCar class.

Q: Can I override properties and use super with a property name to get access to properties in the superclass?

A: No, this only works with methods. You can override properties in a subclass, but you can't use super.propertyName to access a property defined in the superclass.

Q: When you override a method from a superclass, do you always need to call super.[that method]?

A: No. You only need to use super to call the superclass's method if you want to reuse that method's behavior, like we did in RocketCar. Otherwise, you can define entirely new behavior for your subclass.

You've mentioned how messy all those constructor parameters are, and that we have a way to clean them up?

Would you believe, object literals? We've had some discussion of classes versus object literals, and mentioned that object literals are still quite useful, but you really haven't seen a good example of that. Well, the Car class's constructor function is a perfect example of where by using an object literal we can clean up our code a bit and make it more readable and maintainable.

Let's take a look at that now...

Using an object literal to clean up our constructor

Let's look at the Car class again and see how we might be able to clean it up a bit:

Notice we're using a lot of parameters here. We count seven.

The more we add (and we always end up adding more as the requirements for objects grow), the harder this is to read.

```
class Car {
    constructor(make, model, year, color, passengers, convertible, mileage) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
        this.passengers = passengers;
        this.convertible = convertible;
        this.mileage = mileage;
        this.started = false;
    }
    start() {
        this.started = true;
    }
    // rest of the methods here
}
```

And when we write code that makes use of this Car class, we have to make sure we get the arguments all in exactly the right order.

Creating the object literal for the cadi parameters

So, the problem we're highlighting here is that we have a heck of a lot of parameters in the Car class's constructor function, making it difficult to read and maintain. While that might seem like a minor inconvenience, it actually causes more bugs than you might think—and not only that, but they're often nasty bugs that are hard to diagnose at first.

They're hard to diagnose because if you switch two variables, the code is still syntactically correct, but it doesn't function correctly because you've switched two values.

However, there is a common technique that we can use when passing all these arguments that can be used for any function, whether or not it's a constructor function in a class. The technique works like this: take all your arguments, throw them in an object literal, and then pass that literal to your function—that way, you're passing all your values in one container (the literal object) and you don't have worry about matching the order of your arguments and parameters.

Or, if you leave out a value, all kinds of craziness can ensue!

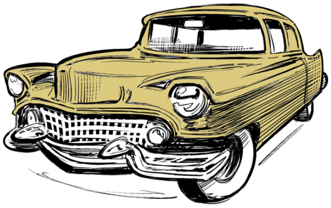
Let's rewrite the code to call Car's constructor function, and then do a slight rework of the constructor code to see how this works. Let's start by taking the call to Car's constructor function and reworking its arguments into an object literal:

All you need to do is take each argument and place it in an object literal with an appropriate property name. We use the same property names used in the constructor.

```
let cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
```

```
let cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};
```

We've kept the same order, but there is no reason you'd have to.



And then we can rewrite the call to the Car constructor like this:

```
let cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};

let cadi = new Car(cadiParams);
```

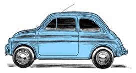
Wow, talk about a makeover. Not only is this much cleaner, it's a lot more readable, at least in our humble opinion.

Now we're passing a single argument to the Car constructor.

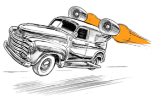
But we're not done yet, because the Car constructor is still expecting seven arguments, not one object. Let's rework that code, and then we'll give this a test.

Exercise

Go ahead and create object literals for the parameters for the other cars too: the Fiat, Chevy, Taxi, and RocketCar.



Your object literals here.



Solution on page 591

Reworking the Car constructor

Now we need to remove all the individual parameters in the Car constructor and replace them with properties from the object that we're passing in. We'll call that parameter `params`. We also need to rework the code a bit to use this object. Here's how:

First things first. We'll replace the seven parameters of the Car constructor with one parameter, for the object we're passing in.

Then, for each reference to a parameter, we substitute the corresponding property from the object passed into the function.

In our methods, we never use a parameter directly. It wouldn't make sense to, because we always want to use the object's properties (which we do using the this variable). So, no changes are needed to this code at all.

```
class Car {
    constructor(params) {
        this.make = params.make;
        this.model = params.model;
        this.year = params.year;
        this.color = params.color;
        this.passengers = params.passengers;
        this.convertible = params.convertible;
        this.mileage = params.mileage;
        this.started = false;
    }
    start() {
        this.started = true;
    }
    stop() {
        this.started = false;
    }
    drive() {
        if (this.started) {
            console.log(`${this.make} ${this.model} goes zoom zoom!`);
        } else {
            console.log("Start the engine first.");
        }
    }
    brake() {
        console.log(`The ${this.make} ${this.model} is braking`);
    }
}

let cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};

let cadi = new Car(cadiParams);
```

No changes here, we've just reproduced the object literal and the call to the Car constructor from the previous page.

Test Drive

Update the cadi and all your other cars, and test your code.

```
cadi.start();
cadi.drive();
cadi.brake();
cadi.stop();
```

Accessor properties

Webville Motors wants to expand its business to the European market, so we need a way to get the mileage of a car in kilometers (kms) rather than miles. What's the best way to do that?

```
class Car {
    constructor(params) {
        this.make = params.make;
        this.model = params.model;
        this.year = params.year;
        this.color = params.color;
        this.passengers = params.passengers;
        this.convertible = params.convertible;
        this.mileage = params.mileage;
        this.started = false;
    }
    // rest of the class here
}
```

We're already storing the miles on the car (in miles, obviously). We'd like a way to get the miles in kms.

We could use two separate properties, but then we'd have to keep track of both miles and kilometers. Or, we could write a method, `getKm`, that takes the current mileage of the car, converts the mileage to kilometers, and returns the kilometers...but then we'd have to remember to access the miles via the property and the kms via the method. Hmmm...

There is a better way. So far, all the properties in our objects have been *data properties*. Data properties are all specified using a property name and value, like we've done so far in object literals, and like we are doing in the Car class above.

JavaScript objects have another kind of property, an *accessor property*. An accessor property is defined using a special method, but the property is accessed as if it were a regular (data) property. To create a *getter* method for an accessor property (that is, a way to *get* the value of the property), we use the keyword **get** in front of a method that returns a value for the property, like this:

```
class Car {
    constructor(params) {
        // rest of the properties here
        this.mileage = params.mileage;
        this.started = false;
    }
    get km() { return this.mileage * 1.60934; }
    // rest of the class here
}
```

Here, we've added a new getter method that returns the current number of miles on the car, converted to kilometers. The get keyword means this is a getter method.

Now that we've defined a getter method, we can use it to get the value of the `km` accessor property. Turn the page to see how...

Using getters

We have a getter method, `km`, in our Car class now, so let's use it to get the kilometers that a Car object has been driven. To use an accessor property, we write that property exactly as if it were a regular (data) property, but behind the scenes the getter method is called. So, to get the value of the `km` accessor property of a `fiat`, we'd write:

```
fiat.km
```

No () needed!

We do not use parentheses to get the value of an accessor property, even though that property is defined using a method. Because of that special keyword **`get`**, we can treat `km` like a regular property.

Here's what this looks like in our code. We'll create a `fiat` object from the Car class with the `km` accessor property and getter method, and then use the `km` property:

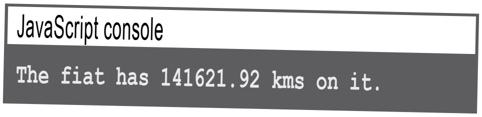
```
class Car {
    constructor(params) {
        this.make = params.make;
        this.model = params.model;
        this.year = params.year;
        this.color = params.color;
        this.passengers = params.passengers;
        this.convertible = params.convertible;
        this.mileage = params.mileage;
        this.started = false;
    }
    get km() { return this.mileage * 1.60934; }
    // rest of the class here
}
let fiatParams = {
    make: "Fiat", model: "500", year: 1957, color: "Medium Blue",
    passengers: 2, convertible: false, mileage: 88000
};
let fiat = new Car(fiatParams);

console.log(`The fiat has ${ fiat.km } kms on it.`);
```

First we create a fiat car by creating an object literal for the parameters, and then passing that to the Car class.

And here's where we use the km getter method. Remember to write the name of the property, and not the ().

Test Drive

Update the fiat, and test your code. Make sure your result matches ours.



```
The fiat has 141621.92 kms on it.
```

What's a getter without a setter?

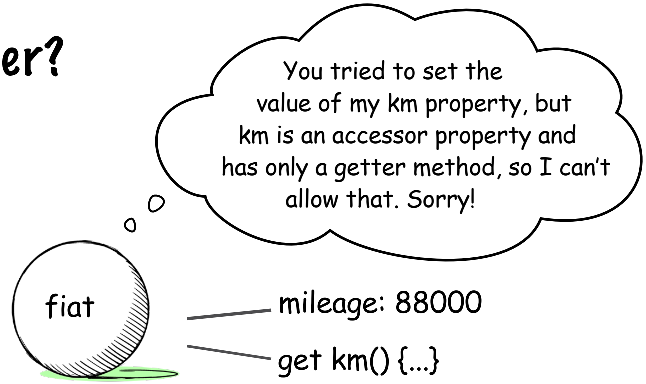
With only a getter method, an accessor property is read-only:

With just a getter method, the `km` property is read-only. You can't set the value of the `km` property to a new value. If you try to assign a new value to the property, like this:

```
fiat.km = 141641;
```

And note, you will not see an error if you try this!

nothing will change.


You tried to set the value of my km property, but km is an accessor property and has only a getter method, so I can't allow that. Sorry!
fiat
mileage: 88000
get km() {...}

Of course, the fiat's `mileage` property is a regular data property, so we can change that no problem:

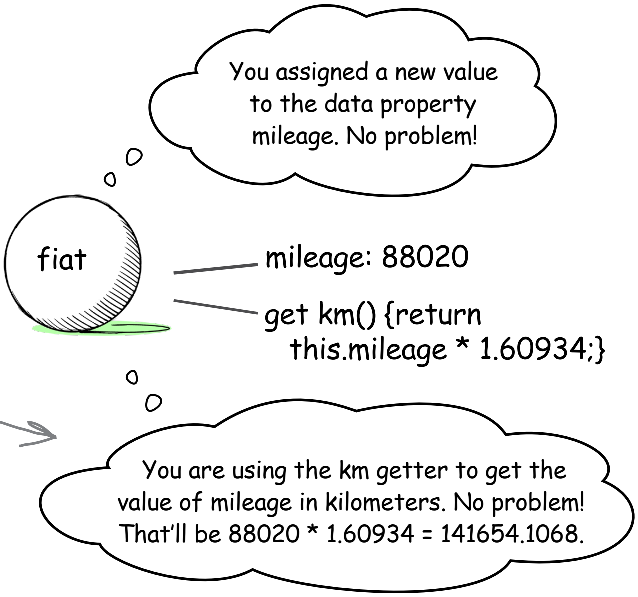
```
fiat.mileage = 88020;
```

This is 141,641 kms.

And then the next time we get the kilometers, we'll get the updated value since the value of the `km` property is always computed by the getter method dynamically:

```
console.log(fiat.km);
```

Accessing the km getter method to compute the kilometers from the new value of mileage works.


You assigned a new value to the data property mileage. No problem!
fiat
mileage: 88020
get km() {return
this.mileage * 1.60934;}
You are using the km getter to get the value of mileage in kilometers. No problem! That'll be 88020 * 1.60934 = 141654.1068.

So, we could update the fiat using the mileage property each time, but that's not as convenient for the European driver.

To set the value of an accessor property, we need a setter method:

To update the mileage of the fiat using kilometers, we can create a setter method for the `km` accessor property. We use the keyword **set**, and create a method with one parameter, the new value for the property:

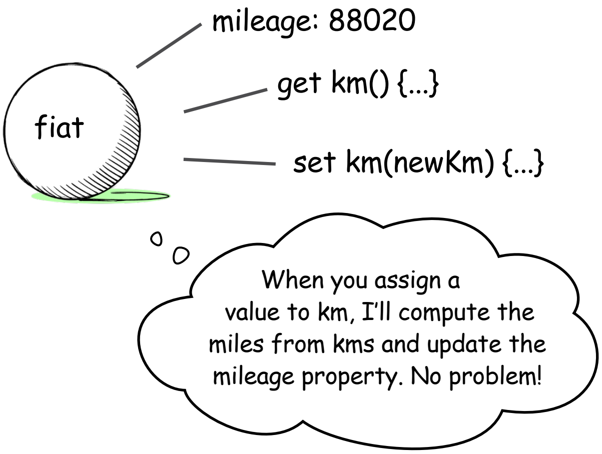
```
set km(newKm) {
    this.mileage = newKm / 1.60934;
}
```

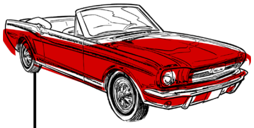
We don't store the kilometers value directly; we convert it to miles and update the mileage property. The mileage property always holds the current number of miles on the car. To set the new mileage of the car using `km`, we write:

```
fiat.km = 141641;
```

This is 88020 miles.

This time it works!


mileage: 88020
get km() {...}
fiat
set km(newKm) {...}
When you assign a value to km, I'll compute the miles from kms and update the mileage property. No problem!



Test Drive

Add the setter method to the fiat, and test your Car class and fiat. The complete code is below.

```
class Car {
    constructor(params) {
        // rest of the setup with params here
        this.mileage = params.mileage;
        this.started = false;
    }
    get km() { return this.mileage * 1.60934; }
    set km(newKm) { this.mileage = newKm / 1.60934; }
    // other methods here
}

let fiatParams = {
    make: "Fiat", model: "500", year: 1957, color: "Medium Blue",
    passengers: 2, convertible: false, mileage: 88000
};
let fiat = new Car(fiatParams);
console.log(`The fiat has ${ fiat.km } kms on it.`);

fiat.start();
fiat.drive();
fiat.stop();
fiat.km = 141641;

console.log(`The fiat has ${ fiat.km } kms on it.`);
console.log(`The fiat has ${ fiat.mileage } miles on it.`);
```

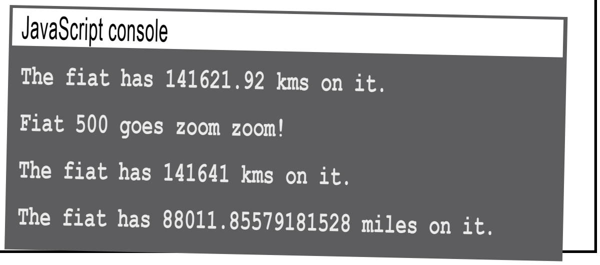
Make sure you add the new setter method for km to your code. This makes km a read/write property.

The fiat has 88000 miles on it, which is 141621.92 kms.

We start the car and drive it 20 kms.

We've finished driving, so we stop the car and now we need to update the mileage. We can do that in Europe by setting a new value for the km property.

We can print out both the value of the km property and the value of the mileage property. We see that mileage got updated correctly using our setter method.



JavaScript console

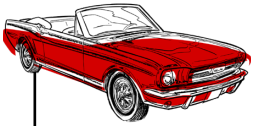
```
The fiat has 141621.92 kms on it.
Fiat 500 goes zoom zoom!
The fiat has 141641 kms on it.
The fiat has 88011.85579181528 miles on it.
```

Can we use getters and setters in object literals too, not just in classes?

Yes, you can. And it's easy. All you have to do is add the keyword **`get`** in front of an object's method to get an accessor property and the keyword **`set`** in front of an object's method to set the value of an accessor property, just like you do in a class. Let's dig out our old code for `fiat` (the code that uses an object literal to create the `fiat` object) and add the `km` accessor property:

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,
    get km() { return this.mileage * 1.60934; },
    set km(newKm) {
      this.mileage = newKm / 1.60934;
    },
    start() {
      this.started = true;
    },
    stop() {
      this.started = false;
    },
    drive() {
      if (this.started) {
        console.log(`${this.make} ${this.model}
goes zoom zoom!`);
      } else {
        console.log("Start the engine first.");
      }
    }
};
```

Note that just like with the other methods in this object literal, we separate each property with a comma (unlike in a class, which does NOT use commas between methods).



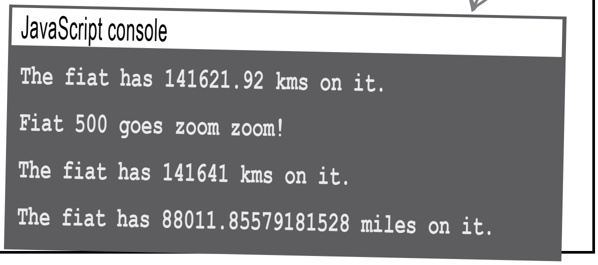
Test Drive

To test the getter and setter methods for a km accessor property in your fiat object literal, dig up your old code for fiat (or copy from it below). Add the getter and setter methods, and test your code. Your result should be exactly the same as in the previous Test Drive, when you used the Car class to create the fiat.

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,
    get km() { return this.mileage * 1.60934; },
    set km(newKm) { this.mileage = newKm / 1.60934; },
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            console.log(`${this.make} ${this.model} goes zoom zoom!`);
        } else {
            console.log("Start the engine first.");
        }
    }
};
console.log(`The fiat has ${ fiat.km } kms on it.`);

fiat.start();
fiat.drive();
fiat.stop();
fiat.km = 141641;

console.log(`The fiat has ${ fiat.km } kms on it.`);
console.log(`The fiat has ${ fiat.mileage } miles on it.`);
```

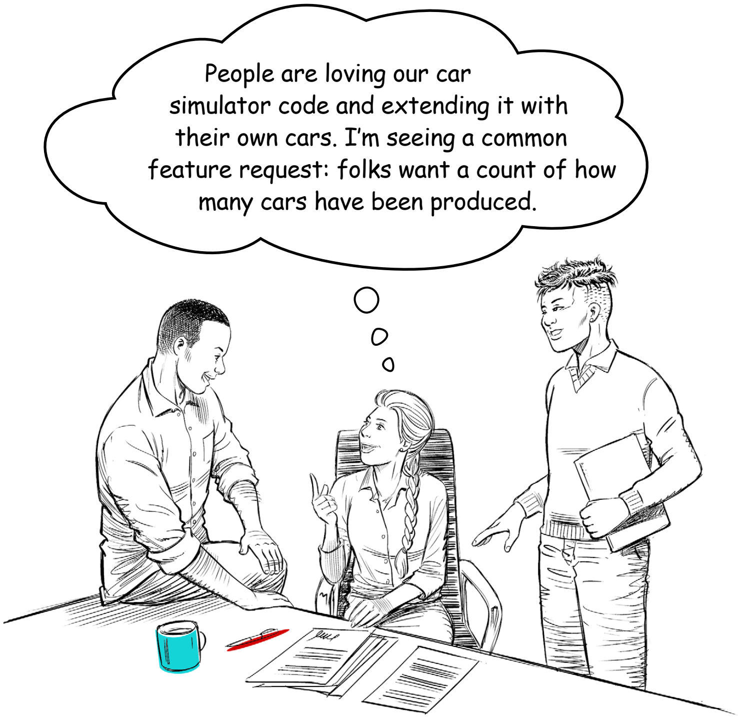
Add the getter and setter methods for km to your fiat object literal. The rest of the code is the same.

Use the same test code as we used for the fiat made from the Car class. You should get exactly the same results.



JavaScript console

```
The fiat has 141621.92 kms on it.
Fiat 500 goes zoom zoom!
The fiat has 141641 kms on it.
The fiat has 88011.85579181528 miles on it.
```


People are loving our car simulator code and extending it with their own cars. I'm seeing a common feature request: folks want a count of how many cars have been produced.

Sam: You mean a count of all the cars that have been instantiated using the `new` operator?

Mara: Exactly.

Alex: That makes sense, but can't we just have a global counter that we always increment when a car is created?

Mara: Well, what if someone forgets to increment it? Or someone even decrements it? From a code perspective, that doesn't sound very safe.

Sam: I'm trying to think of how we could do that. When I create a new object, all the properties in that object belong, well, to the object. A counter is pointless if we make it part of the the car objects we create from Car. Every counter will always be 1!

Mara: Exactly my thinking. So what if we make it part of the class, not the object?

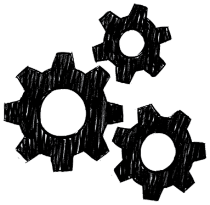
Alex: Wait, what is this magic you speak of? Part of the class??

Mara: It is a bit like magic, Alex. It's called a **static** property. We can make static methods too, if we want. Because there is only one class, there will be only one `counter` property.

Sam: Sounds very promising.

Mara: I think so, and I think this will be straightforward. Let's give it a try and see if we can quickly get this new feature out to folks.

Static properties and methods



Static properties and methods are similar to object properties and methods, except they belong to a class. For properties, that means there is only one property no matter how many objects you instantiate from a class, because there is only one class. For methods, that means we can call a method on a class without having to have an object instantiated. To add a static property or method, we use the **`static`** keyword:

```
class Widget {
    static counter = 0;


    constructor(type) {
        this.type = type;
        Widget.counter++;
    }


    static getCounter() {
        return Widget.counter;
    }
}
```

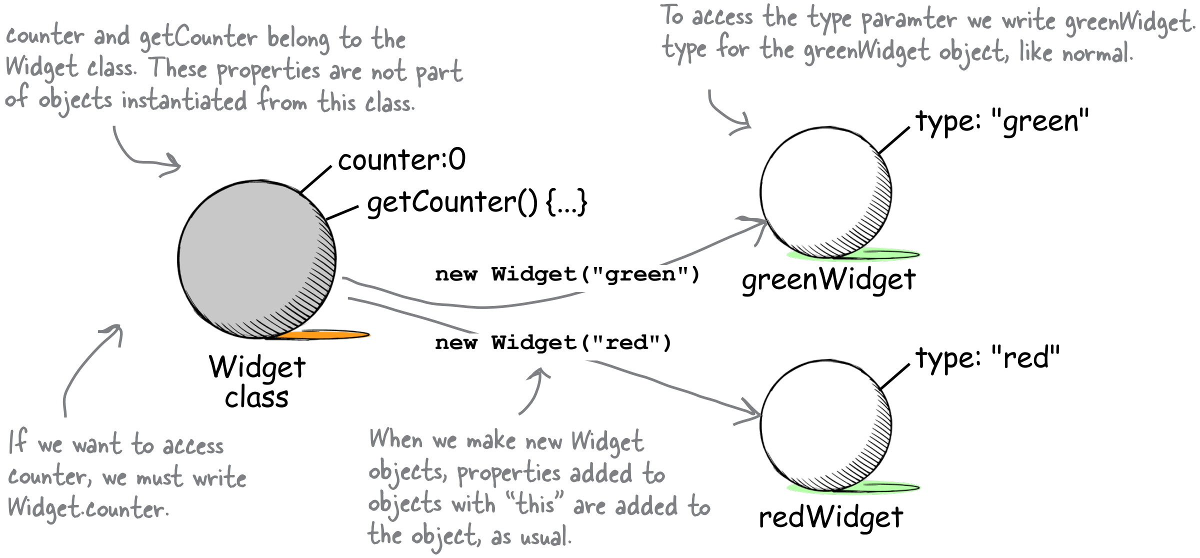
Here we define a static property, counter, and initialize it to 0.

In the constructor, we have one parameter that we use to set a property on all objects instantiated from the class.

Each time we make a new object from Widget, we'll increment counter. But note: we must use the name of the class to access the counter property.

And here we have a static method to get the value of the counter. This is a method of the Widget class. We return the value of counter, and again, we must use the class name to access the counter property.

In the Widget class, we have a static property, `counter`, and a static method, `getCounter`. The `counter` property is a property of the class, and the `getCounter` method is a method of the class. Neither of these are added to objects that we instantiate from Widget.

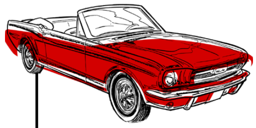


counter and getCounter belong to the Widget class. These properties are not part of objects instantiated from this class.

To access the type paramter we write greenWidget.type for the greenWidget object, like normal.

If we want to access counter, we must write Widget.counter.

When we make new Widget objects, properties added to objects with "this" are added to the object, as usual.



Test Drive

Let's test our Widget class. We'll test the static property and method, and we'll make some objects from the Widget class and test those too. Time to experiment!

We've set up some code to test how static properties and methods work.

First let's print the value of counter, which should be 0. Note we're using the class name to access the static property, counter.

```
console.log("Counter:", Widget.counter);
```

Now let's try creating a new Widget with the type "green". When we create a new Widget, the counter property should get incremented, so if we print the counter value again, it should now be 1.

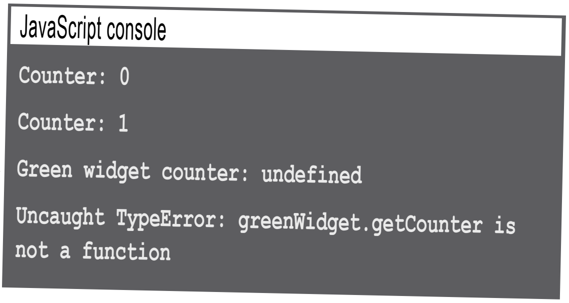
```
let greenWidget = new Widget("green");
console.log("Counter:", Widget.getCounter());
```

As an experiment, let's try accessing counter from our greenWidget and see what happens. We'll try to print the value of counter directly, and we'll try using the getCounter method.

```
console.log("Green widget counter:", greenWidget.counter);
console.log("Green widget getCounter():", greenWidget.getCounter());
```

We can see here that accessing counter using the class works great, and also that our counter got updated properly when we made a new object from the Widget class.

And we can see that if we try to access counter from greenWidget, it's undefined because counter isn't a property of greenWidget, it's a property of the class. If we try to call the getCounter method, we get an error.

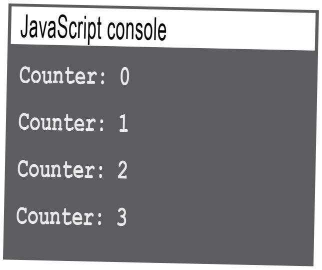


JavaScript console

```
Counter: 0
Counter: 1
Green widget counter: undefined
Uncaught TypeError: greenWidget.getCounter is
not a function
```

```
console.log("Counter:", Widget.counter);

let greenWidget = new Widget("green");
console.log("Counter:", Widget.getCounter());

let blueWidget = new Widget("blue");
console.log("Counter:", Widget.getCounter());

let redWidget = new Widget("red");
console.log("Counter:", Widget.getCounter());
```

Erase those last two lines and add two more widgets to make sure counter is working as it should. Ours is! Each time we add a new Widget, the counter is incremented. Cool!



JavaScript console

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
```

Counting our car production

We can use the exact same idea as we just used for counting widgets to count cars. We'll just add a static `counter` property to the Car class, along with a static `getCounter` method to get the current counter value. Looks like someone is thinking ahead and has added a static `resetCounter` method in case we ever want to reset the counter:

```
class Car {
    static counter = 0;
    constructor(params) {
        this.make = params.make;
        this.model = params.model;
        this.year = params.year;
        this.color = params.color;
        this.passengers = params.passengers;
        this.convertible = params.convertible;
        this.mileage = params.mileage;
        this.started = false;
        Car.counter++;
    }
    static getCounter() { return Car.counter; }
    static resetCounter() { Car.counter = 0; }
    // rest of the Car methods here
}
```

We've added the static counter property here, and initialized it to 0.

We're incrementing the counter in the Car constructor, so every time we make a new Car, we'll add one to the counter.

And we've got two static methods, to get the current value of counter and to reset the counter to 0 in case we need to.

Sam: I got this! Check out my code. I added the static `counter` property and a static method to get the current value of the counter. I even threw in a `resetCounter` method in case we want to do a "factory reset."

Mara: Good thinking Sam, and great job.

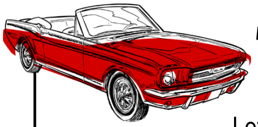
Alex: Wait, don't we need to add the counter to Taxi and RocketCar too?

Mara: No...remember, Taxi and RocketCar extend the Car class. In our Taxi and RocketCar constructors, we're calling `super(...)`, which calls the Car constructor, and that's where the counter is incremented.

Alex: Ohhhh, that makes sense. So if we forgot to call `super(...)` in the Taxi and RocketCar constructors, we'd have a problem, right?

Mara: Right. But we're calling `super(...)`, and typically that's what we do when we extend a class, so we'll have no problem keeping the counter accurate for Cars, Taxis, and RocketCars.

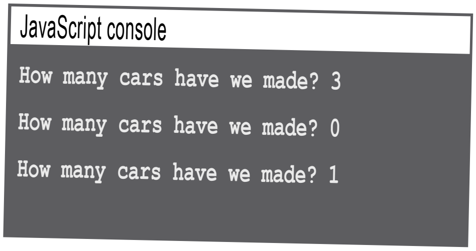
Sam: See, told ya. I got this!



Test Drive

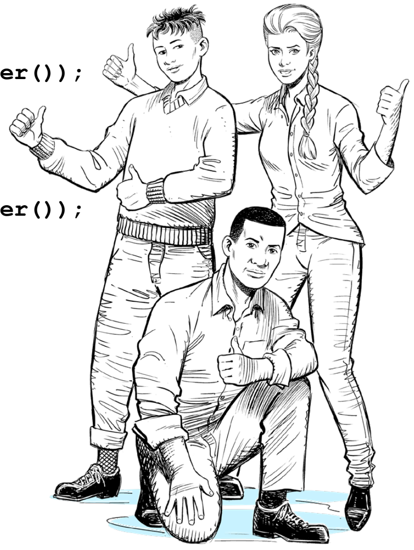
Let’s test our car production counter. We want to make sure it counts Cars, Taxis, and RocketCars correctly.

```
let fiatParams = {
    make: "Fiat", model: "500", year: 1957, color: "Medium Blue",
    passengers: 2, convertible: false, mileage: 88000
};
let fiat = new Car(fiatParams);

let taxiParams = {
    make: "Webville Motors", model: "Taxi", year: 1961, color: "Yellow",
    passengers: 3, convertible: false, mileage: 7340, fare: 2.50
};
let taxi = new Taxi(taxiParams);

let rocketCarParams = {
    make: "Estes", model: "Centurion", year: 1963, color: "Blue",
    passengers: 0, convertible: false, mileage: 300000000
};
let rocketCar = new RocketCar(rocketCarParams);

console.log("How many cars have we made?", Car.getCounter());

// use only if needed to do a "factory reset"!
Car.resetCounter();
console.log("How many cars have we made?", Car.getCounter());

// remake the rocketCar
rocketCar = new RocketCar(rocketCarParams);
console.log("How many cars have we made?", Car.getCounter());
```



JavaScript console

```
How many cars have we made? 3
How many cars have we made? 0
How many cars have we made? 1
```

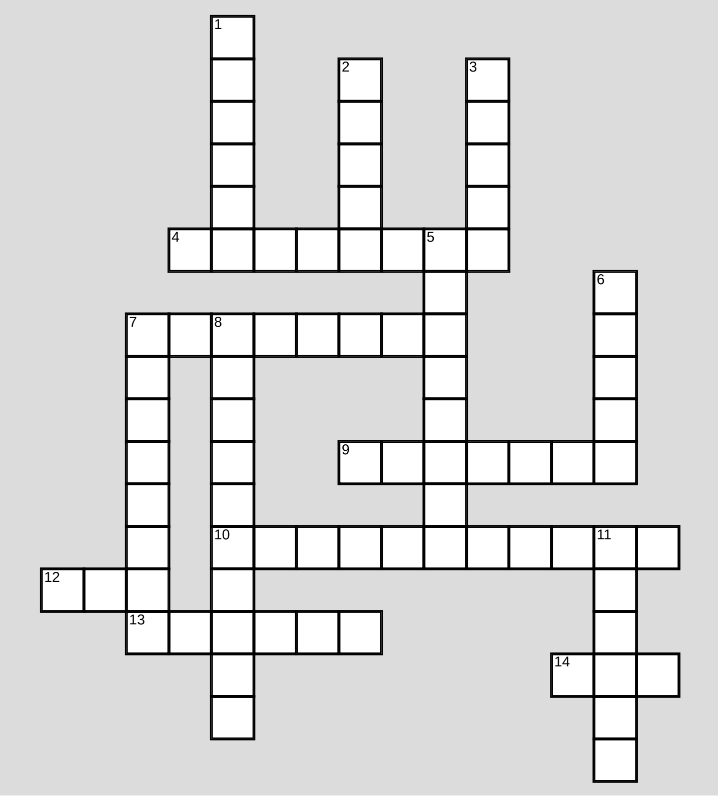


Bullet Points

- An **object literal** works well when you need to create a small number of objects.
- A **class** works well when you need to create many similar objects.
- We capitalize the first letter of class names by convention.
- Using a class, we can create objects that are consistent, having the same property names and methods.
- Use the **new** operator with a class to call the class's constructor function and create an object.
- When you use new with a class, it creates a new, empty object, which is assigned to **this** within the body of the constructor function.
- Use **this** in a class's constructor function to access the object being constructed and add properties to the object.
- A new object is returned automatically by the class's constructor function.
- If you forget to use **new** with a class, no object is created. This will cause errors in your code that can be difficult to debug.
- To customize objects, we pass arguments to the class's constructor function and use those values to initialize the properties of the object being created.
- If a class's constructor function has a lot of parameters, consider consolidating them into one object parameter.
- Using the **extends** keyword, we can extend one class with another.
- When you extend a class, the class being extended is called the **superclass**, and the class doing the extending is called the **subclass**.
- When we extend a class, the subclass **inherits** properties and methods form the superclass.
- The **super** keyword allows us to call the constructor and other methods in the superclass.
- You may **override** methods and properties from the superclass in the subclass.
- Use the **set** and **get** keywords to create **accessor properties**: setter and getter methods that are treated like properties.
- **Static** properties belong to a class. If you define a static property in a class, there is only one of them, no matter how many objects you make from that class.
- Static methods also belong to a class. To invoke them you need to invoke them on the class, not an instance of the class.

JavaScript cross

Construct some new connections in your brain with this crossword puzzle.



ACROSS

4. Use the set keyword to make a writable _______ property.

7. When you create an object from a class, we say the object is an _______ of the class.

9. Classes define both properties and _______.

10. Use the _______ function in a class to assign properties to an object.

12. Use the _______ keyword to make a readable accessor property.

13. _______ properties belong to a class, not an object.

14. Class and _______ work well as a team.

→ Solution on page 592

DOWN

1. We counted cars from the Car class with a _______ property.

2. Use a _______ to create many similar objects.

3. You can call the constructor or another method of the superclass using this keyword.

5. To create an object with a class, you use the new _______.

6. To access a static property or method, use the _______ name.

7. When you extend a class, the subclass _______ methods and properties from the superclass.

8. The subclass extends a _______.

11. The class's constructor function returns the newly created _______.

From page 552

BE the Browser Solution

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors. Here's our solution.

If widget is to be a constructor, it should start with a capital W. This won't cause an error, but it's a good convention to follow.

We don't need "let" in front of this. We're not declaring new variables, we're adding properties to an object.

```
function widget(partNo, size) {
    let this.no = partNo;
    let this.breed = size;
}
```

Also, by convention we usually give the parameters the same names as the properties, so this.partNo and this.size would probably be better.

We're using commas instead of semicolons. Remember, in the constructor we use normal statements rather than comma-separated property name/value pairs.

```
function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}
```

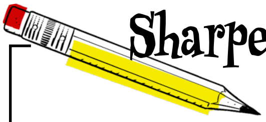
We're returning this, but we don't need to. The constructor will do it for us. This statement won't cause an error, but it's not necessary.

Forgot new!

Needs a space between new and the constructor name.

```
let widgetA = widget(100, "large");
let widgetB = new widget(101, "small");
let formFactorA = newFormFactor("plastic", widgetA);
let formFactorB = new ForumFactor("metal", widgetB);
```

Misspelled the name of the constructor.



Sharpen your pencil Solution

From page 549

Do a little pattern matching and see if you can write a class for creating Duck objects. You'll find one of our duck object literals below to base your class on. Here's our solution.

```
let duck = {
    type: "redheaded",
    canFly: true
}
```

Here's an example duck object literal.

Write a constructor for creating Ducks.

```
class Duck {
  constructor(type, canFly) {
    this.type = type;
    this.canFly = canFly;
  }
}
```

Sharpen your pencil Solution

From page 555

Let's make sure you've got the subtleties of object literals and classes down. You'll find a Dog object literal on the top, and a class on the bottom. Compare and contrast. In what ways are they alike and different? Here's our solution.

Let's start with an object literal.

Remember that with an object literal we're making one and only one object. So if we want more dogs, we have to make more object literals.

```
let fido = {
    name: "Fido",
    breed: "Mixed",
    weight: 38,
    bark() {
        if (this.weight > 25) {
            alert(`${this.name} says Woof!`);
        } else {
            alert(`${this.name} says Yip!`);
        }
    }
};
```

Object literals are defined with properties consisting of a name and value, separated by a colon.

Each property is separated by a comma.

As we saw earlier, we can use the method shorthand in objects, and this looks more like how we define methods in classes.

We assign the object literal, delimited with { and }, to a variable and end the assignment statement with a semicolon.

Now let's look at a class.

With a class, we're not defining a dog object, we're creating a class we can use to make multiple Dog objects.

```
class Dog {
    constructor(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
    }
    bark() {
        if (this.weight > 25) {
            alert(`${this.name} says Woof!`);
        } else {
            alert(`${this.name} says Yip!`);
        }
    }
}

let fido = new Dog("Fido", "Mixed", 38);
```

All objects we make from class Dog will have the same properties and methods.

When we write "new Dog(...)", the constructor function assigns property values to the newly created dog object. The resulting dog object has exactly the same name, breed, and weight property names as the fido object above has.

No comma, no semicolon here!

We do NOT use the function keyword here.

We add methods to a class (and to the objects we make from classes) by specifying them in the class like this.

If we had more methods to add, we'd add them here. Again, no comma, no semicolon.

No semicolon here either.

Finally, we create the fido object using the Dog class and new, like this.

Exercise Solution

From page 557

We've got a constructor to create coffee drinks, but it's missing its methods.

We need a method, getSize, that returns a string depending on the number of ounces of coffee:

- 8 oz is a small
- 12 oz is a medium
- 16 oz is a large

We also need a method, toString, that returns a string that represents your order, like "You've ordered a small House Blend coffee."

Write your code below, and then test it in the browser. Try creating a few different sizes of coffee. Here's our solution.

```
class Coffee {
    constructor(roast, ounces) {
        this.roast = roast;
        this.ounces = ounces;
    }
    getSize() {
        if (this.ounces === 8) {
            return "small";
        } else if (this.ounces === 12) {
            return "medium";
        } else if (this.ounces === 16) {
            return "large";
        }
    }
    toString() {
        return (`You've ordered a ${this.getSize()} ${this.roast} coffee.`);
    }
}
```

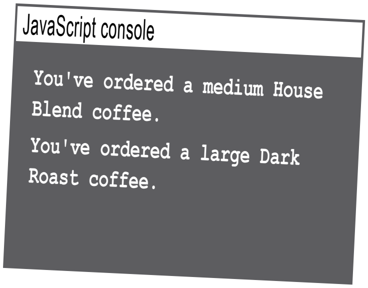
The getSize method looks at the ounces property of the object and returns the corresponding size string.

Remember, this will be the object whose method we call. So if we call houseBlend.getSize, then this will be the houseBlend object.

The toString method just returns a string description of the object. It uses the getSize method to get the size of the coffee.

We create two coffee objects, call the toString method, and display the resulting string.

```
let houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());

let darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```



```
You've ordered a medium House
Blend coffee.
You've ordered a large Dark
Roast coffee.
```

Here's our output; yours should look similar.

Exercise Solution

From page 566

Why don't you take a crack at the code for the RocketCar constructor? Like Taxi, RocketCar will extend the Car class. It needs just one extra property, thrustOn, which will be initialized to false. Here's our solution.

This code is just like the Car constructor.

```
constructor(make, model, year, color, passengers, convertible, mileage) {
    super(make, model, year, color, passengers, convertible, mileage);
    this.thrustOn = false;
}
```

We need to call super to take advantage of all the initialization code in Car's constructor.

And we also need to add and set the value of the thrustOn property to false.

Exercise Solution

From page 573

Go ahead and create object literals for the parameters for the other cars too: the Fiat, Chevy, Taxi, and RocketCar.

Here's our solution.

```
let fiatParams = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};
```

```
let chevyParams = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

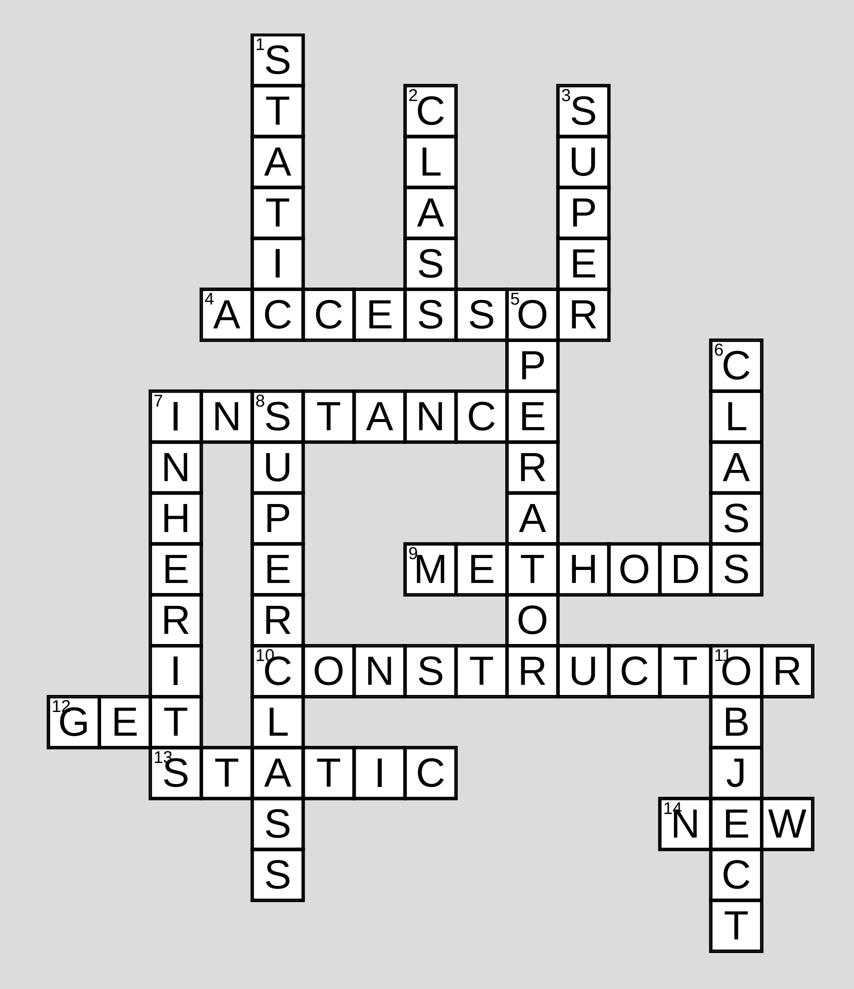
```
let taxiParams = {
   make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

```
let rocketCarParams = {
    make: "Estes",
    model: "Centurion",
    year: 1963,
    color: "blue",
    passengers: 0,
    convertible: false,
    mileage: 300000000
};
```

JavaScript cross Solution

From page 587

Construct some new connections in your brain with this crossword puzzle. Here's our solution.


1 STATIC
2 CLASS
3 SUPER
4 ACCESSOR
5 OPERATOR
6 CLASS
7 INSTANCE
7 INHERIT
8 SUPERCLASS
9 METHODS
10 CONSTRUCTOR
11 OBJECT
12 GET
13 STATIC
14 NEW

Appendix: leftovers

The top ten topics (we didn't cover)

jelly
Almond
MILK

We've covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small chapter. Actually, we *did* originally include everything you need to know about JavaScript programming (not already covered by the other chapters), by reducing the type point size to .00004. It all fit, but nobody could read it. So we threw most of it away, and kept the best bits for this "top ten" appendix.
This really *is* the end of the book. Except for the index, of course (a must-read!).

#1 Modules

We talked a lot about scope in this book. One JavaScript language feature related to scope that we didn't mention is modules. A **module** is another way to limit the visibility of your code, at the file level.

Using modules is easy. You just need to learn two new keywords and a few simple rules. Let's say you have a file, "taxi.js", containing a class `Taxi` and a function `createFleet` that creates a fleet of taxis. You also have a web page that creates a list of taxis for users who are interested in hailing a taxi.

You could link from your HTML to your "taxi.js" file using a regular script link, like this:

```
<script src="./taxi.js"></script>
<script>
  // code to create and list a fleet of taxis
</script>
```

But while you'd like folks to be able to call `createFleet`, you don't want them instantiating taxis directly. With a module, you can control what is exported from your "taxi.js" file using the `export` keyword. Let's look at the "taxi.js" file that contains the `Taxi` class and the `createFleet` function to see how this works:

```
class Taxi {
    constructor(model, medallion) {
        this.model = model;
        this.medallion = medallion;
        this.miles = 0;
    }
    trip(miles) {
        this.miles = this.miles + miles;
    }
}

export function createFleet(fleetSize = 5, medallion = 0) {
    let fleet = [];
    for (let i = 0; i < fleetSize; i++) {
        let taxi = new Taxi("Webville Motors", medallion);
        fleet.push(taxi);
        medallion++;
    }
    return fleet;
}
```

Here we have a slimmed-down version of our Taxi class; it has just three properties and one method.

The export keyword can be used in a module to export functions, variables, even classes. Here, we want only the createFleet function to be accessible by code outside this file.

Notice the export keyword before the function declaration here.

The createFleet function takes the number of taxis in the fleet and the starting medallion number. It creates an array of taxis and returns that array.

The only thing that distinguishes the "taxi.js" module from a regular ".js" file is the use of the export keyword.

To use the module, we must import it into our JavaScript where we want to use it. Let's say we have a web page, "taxi.html", that will display the taxis in the page. Rather than linking to the "taxi.js" file like we're used to (and as shown on the previous page), now we use the **import** keyword in our JavaScript, like this:

Imagine the usual HTML and CSS here.

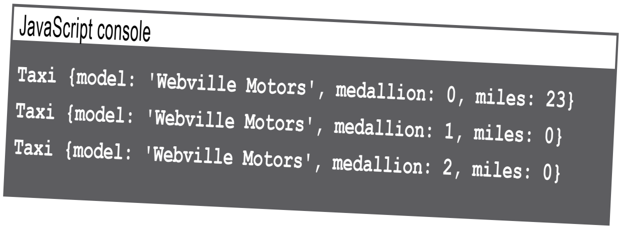
```
<script type="module">

  import { createFleet } from "./taxi.js";

  let fleet = createFleet(3, 0);
  fleet[0].trip(23);
  console.log(fleet);
</script>
```

The first thing to notice is that we can only import a module that has an export into another module. So, we identify the code in the web page with the type "module".

Here, we're importing the JavaScript in the file "./taxi.js", meaning the "taxi.js" file in the same folder as the HTML file is in. But we're not importing ALL of it; we're importing only the createFleet function, because that's the only code being exported. We specify which items we're importing between { } in the import statement.

Once we've imported the function createFleet, we can use it just as if it were right there in our code.

We call it to create a fleet of 3 taxis, starting with medallion number 0. We drive the first taxi 23 miles. Then we print the taxis to the console.

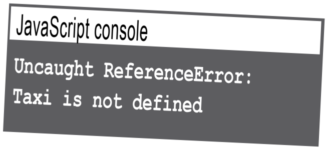


```
Taxi {model: 'Webville Motors', medallion: 0, miles: 23}
Taxi {model: 'Webville Motors', medallion: 1, miles: 0}
Taxi {model: 'Webville Motors', medallion: 2, miles: 0}
```

Note that our code in the "taxi.html" file cannot access the `Taxi` class, because that was neither exported from the "taxi.js" module nor imported into the code in "taxi.html". If you try it, like this:

```
let taxi = new Taxi("Really Great Taxi", 0);
```

you'll see:



```
Uncaught ReferenceError:
Taxi is not defined
```

You can export multiple variables (or functions or classes) from one file if you need to. For instance, if "taxi.js" exports both the `createFleet` function and a `taxiCompany` variable, you can import both into the code in "taxi.html" like this:

```
import { createFleet, taxiCompany } from "./taxi.js";
```

#2 JSON

Not only is JavaScript the programming language of the web, it's also a common interchange format for storing and transmitting objects. JSON, an acronym for "JavaScript Object Notation," is a format that allows you to represent a JavaScript object as a string—a string that can be stored and transmitted:

A JSON string.

```
let fidoString = '{ "name": "Fido", "breed": "Mixed", "weight": 38 }';
```

Notice that we're using single quotes around the JSON string. We have to use single quotes because the string contains double quotes, so JavaScript will get confused otherwise. This way, JavaScript knows this is one long string that contains other strings.

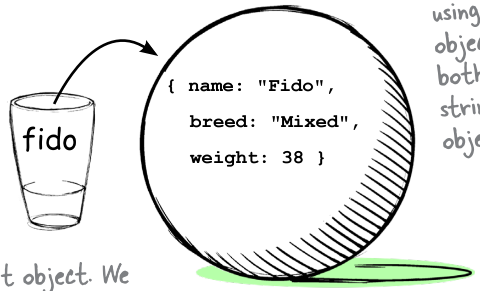
Look familiar? It should. This string looks a lot like the fido object we worked with earlier in the book...

Now, the cool thing about JSON is that we can take strings like this and turn them into objects. We do this with a couple of methods supplied by JavaScript's `JSON` object: `JSON.parse` and `JSON.stringify`. We'll use the `parse` method to parse the `fidoString` above and turn it into a real dog (well, a JavaScript object, anyway):

Notice that we're using the JSON object here. JSON is both the name of a string format and an object in JavaScript.

```
let fido = JSON.parse(fidoString);
```

We call the parse method of the JSON object, passing the string above, and we get back...



...a real JavaScript object. We store the reference to the object in the variable fido.

And you can go the other way, too. If you have an object, `fido`, and you want to turn it into a string, you just call the `JSON.stringify` method, like this:

```
let fido = {
    name: "Fido",
    breed: "Mixed",
    weight: 38
};
let fidoString = JSON.stringify(fido);
```

Here, we're taking a JavaScript object...

...and turning it into a string.

Note that the JSON format doesn't work with methods (so you can't include, say, a `bark` method in your JSON string), but it does work with all the primitive types, as well as objects and arrays.

#3 Promises

Callbacks are a common way to handle asynchronous operations in JavaScript. Overuse of callbacks can result in a large number of nested callbacks that become hard to manage. This is commonly known as *callback hell*. **Promises** provide a more natural way to specify a sequence of asynchronous operations. Promises also provide a means to handle error conditions in asynchronous calls, which can be difficult with callbacks.

A Promise is an object that represents the eventual completion of an asynchronous operation and its resulting value. Let's look at an example using JavaScript's `fetch` function, which you can use to retrieve resources across the network. The `fetch` function, when called with a URL, does not wait for a response from the network; instead, the function returns immediately and hands back a Promise object:

```
let url = "<your favorite URL here>";
let promise = fetch(url);
```

fetch returns a Promise object that acts as a placeholder for a future value.

Rather than passing `fetch` a callback to invoke when `fetch`'s data is returned from the network, we call the `then` method of the Promise returned from `fetch` and pass that `then` method a function:

```
let url = "<your favorite URL here>";
let promise = fetch(url)
    .then(response => response.json())
    .then(data => display(data))
    .catch(error => handle(error));
```

After fetching the data, translate it to JSON format.

Then show it to the user.

If there's an error, let the handle function take care of that.

While this may seem like just another way to specify callbacks, and in some sense it is, with Promises we can take those callbacks and rewrite them in a more linear fashion that is easier to read and reason about.

Promises can be used with any asynchronous code, and you can write your own Promises.

While a simple idea on the surface, Promises can take a while to master. They are powerful and worthy of study, as they are used in many frameworks that undoubtedly lie ahead in your programming future.

#4 Destructuring assignment

You already know you can assign an array to a variable:

```
let colas = ["Cola", "Diet Cola", "Root Beer"];
```

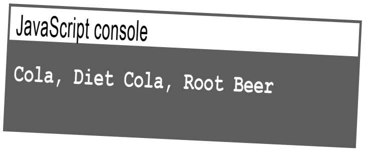
What do you think happens if you put an array on the left side of the assignment?

```
let [cola, diet, rootBeer] = colas;
```

Looks like an array on the left, but it's actually destructuring an array into variables.

You get a **destructured assignment**. In this case, what happens is the array items in the `colas` array are destructured into individual variables, `cola`, `diet`, and `rootBeer`. Let's display them in the console:

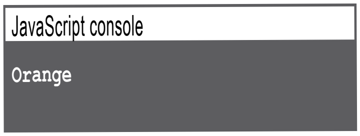
```
console.log(`${cola}, ${diet}, ${rootBeer}`);
```



```
Cola, Diet Cola, Root Beer
```

You can combine destructuring assignment with other syntax you learned in this book, like default values:

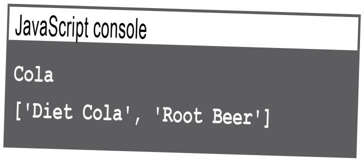
```
let [cola, diet, rootBeer, orange = "Orange"] = colas;
console.log(orange);
```



```
Orange
```

or rest values:

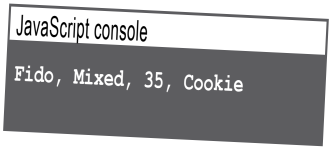
```
let [cola, ...drinks] = colas;
console.log(cola);
console.log(drinks);
```



```
Cola
['Diet Cola', 'Root Beer']
```

You can destructure objects too:

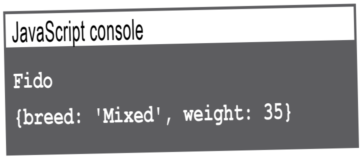
```
let fido = {
    name: "Fido",
    breed: "Mixed",
    weight: 35
};
let {name, breed, weight = 30, handler = "Cookie"} = fido;
console.log(`${name}, ${breed}, ${weight}, ${handler}`);
```



```
Fido, Mixed, 35, Cookie
```

and:

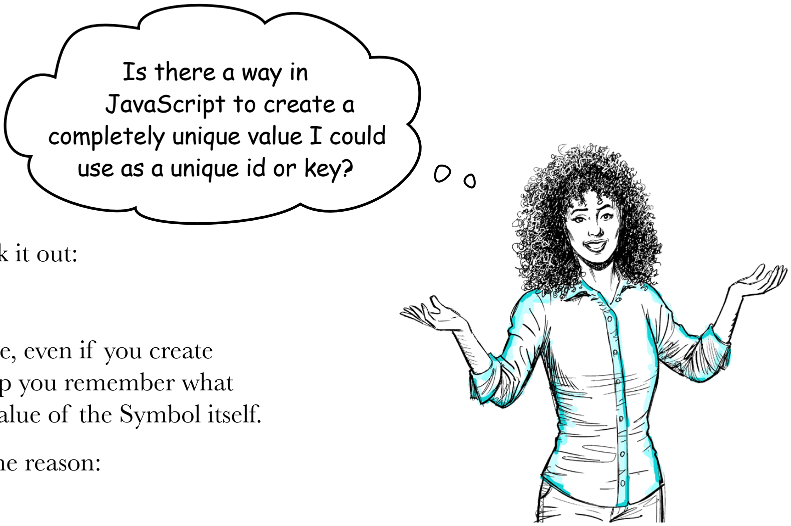
```
let {name, ...restOfDog} = fido;
console.log(name);
console.log(restOfDog);
```



```
Fido
{breed: 'Mixed', weight: 35}
```

In short, using destructured assignment plus default values and the spread and rest operators gives you a lot of flexibility in assigning values to variables in concise code.

#5 Symbols and BigInt


Is there a way in JavaScript to create a completely unique value I could use as a unique id or key?

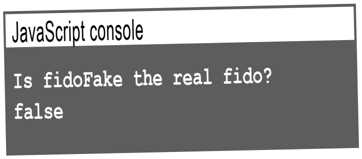
Why, yes there is. The Symbol. Let's check it out:

```
const fidoSym = Symbol("fido");
```

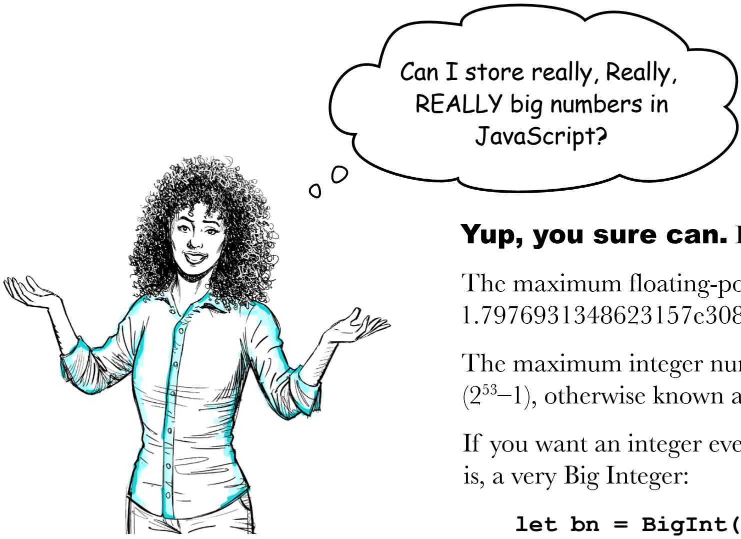
The value in `fidoSym` is guaranteed to be unique, even if you create another one. The "fido" string is just a tag to help you remember what the Symbol is for; it makes no difference to the value of the Symbol itself.

This is handy when you need a unique id for some reason:

```
let fido = {
    name: "Fido",
    sym: Symbol("fido")
};
let fidoFake = {
    name: "Fido",
    sym: Symbol("fido")
}
console.log("Is fidoFake the real fido?", fido.sym == fidoFake.sym);
```


JavaScript console

```
Is fidoFake the real fido?
false
```


Can I store really, Really, REALLY big numbers in JavaScript?

Yup, you sure can. Here are your choices.

The maximum floating-point number you can safely store is 1.7976931348623157e308, otherwise known as `Number.MAX_VALUE`.

The maximum integer number you can safely store is 9007199254740991 (2^{53}–1), otherwise known as `Number.MAX_SAFE_INTEGER`.

If you want an integer even bigger than 2^{53}–1, you can use a BigInt—that is, a very Big Integer:

```
let bn = BigInt(2 ** 309);
console.log(bn);
```

Whoa! Try it!

There is no theoretical limit to the size of a BigInt, but obviously there is a pracical limit.

Note that BigInts don't play well with Numbers, so don't mix and match.

#6 Map and Set

Along with arrays and objects, there are two other useful data structures we didn't touch on at all in this book: **Map** and **Set**. Let's take a brief look at what these are and how to use them, and you can explore them more on your own. We'll begin with Map.

At first glance, a Map might sound just like an object: that is, a Map is a collection of key/value pairs, kind of like an object is a collection of property names and values. But objects are designed to have a fairly stable set of properties—that is, you'll likely not add and remove them frequently. As we saw with cars and dogs, generally your objects are meant to contain properties that describe a thing, and those don't change that often (although the property values might).

Maps are designed specifically so you can add and remove key/value pairs frequently. They don't describe a thing, as much as hold related data that is organized by keys.

Let's take a look at an example to make this more concrete. Imagine you're managing a dog show and you want to keep track of how many of each breed of dog will be in the show. You can use a Map, like this:

```
let breeds = new Map();
breeds.set("Mixed", 2);
breeds.set("Poodle", 2);
breeds.set("Chihuahua", 1);
console.log(breeds.get("Mixed"));
```

First, we make a new, empty Map.

Note: you can repeat values, but not keys. Keys must be unique.

Then we add 3 key/value pairs, using a string for the key and an integer for the value, with the Map's set method.

Finally, we get the value for the "Mixed" key from the Map to print it out, using the Map's get method.

Note that the keys used in a Map must be unique, but you can assign any value to a given key. A key can be a primitive (string, number, boolean, etc.), or even an object. So can values. Maps are very flexible that way:

This is the key. This is the value.

```
breeds.set("Mixed", 2);
```

Together they make a key/value pair.

Let's look at a practical example of how we might use a Map with other data structures to manage the dog show. Here's an array of dogs that will be in the show:

```
let fido = new Dog("Fido", "Mixed", 35);
let spot = new Dog("Spot", "Chihuahua", 10);
let fluffy = new Dog("Fluffy", "Poodle", 30);
let rover = new Dog("Rover", "Mixed", 29);
let max = new Dog("Max", "Poodle", 22);
let frankie = new Dog("Frankie", "Mixed", 38);

let dogsInShow = [ fido, spot, fluffy, rover, max, frankie ];
```

We create a bunch of dogs...

...and add them to an array.

Let's see how we can use a Map to make it easy to figure out how many dogs of each breed we have in the show:

We'll pass in the dogs array to our function, countDogsByBreed.

```
function countDogsByBreed(dogs) {
    const dogBreedMap = new Map();
    dogs.forEach(dog => {
        let count = dogBreedMap.get(dog.breed) || 0;
        dogBreedMap.set(dog.breed, count+1);
    });
    return dogBreedMap;
}
let breedMap = countDogsByBreed(dogsInShow);
```

Create a new Map. We'll use the dog breed string as the key, like before.

For each dog in the array, we'll get the count of that breed in the Map, or if that breed isn't in the Map yet, we'll set count to 0. Then we add the dog's breed and increment count.

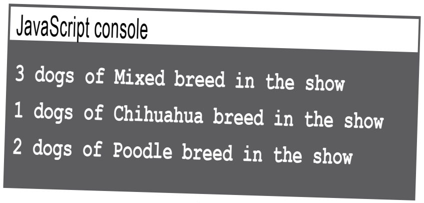
Return the Map.

Here's where we call our function and get back a Map of dog breeds (keys) and counts of those breeds (values).

Now that we have the breeds and the number of each breed in a Map, we can easily iterate over the Map to print out a report. We can use `forEach` with a Map to iterate, like this:

When you use forEach with a Map, the function you pass to forEach has two arguments (not one, like in an array forEach). The first is the value, and the second is the key.

```
breedMap.forEach((count, breed) =>
    console.log(`${count} dogs of ${breed} breed in the show`)
);
```



```
3 dogs of Mixed breed in the show
1 dogs of Chihuahua breed in the show
2 dogs of Poodle breed in the show
```

The Map makes it handy for us to easily manage the breeds in the show. Want to know how many Poodles we have?

```
console.log("Number of poodles in the show:", breedMap.get("Poodle"));
```

Want to know how many different breeds are in the show?

```
console.log("Number of different breeds in the show:", breedMap.size);
```

The Map uses breeds as the keys, and keys are unique, so the size is the number of different breeds.

Want to know if the show includes any Mixed breed dogs?

```
console.log("Does the show include Mixed breed dogs?",
                breedMap.has("Mixed") && breedMap.get("Mixed") > 0);
```

Use the Map's has method to get a true or false value indicating whether "Mixed" is one of the keys in the Map.

If the Map has a key "Mixed", get the value and make sure it's > 0.

Another useful data structure for collecting objects together is a Set. Let's go back to our dogs and the `dogsInShow` array to demonstrate how Set works:

```
let fido = new Dog("Fido", "Mixed", 35);
let spot = new Dog("Spot", "Chihuahua", 10);
let fluffy = new Dog("Fluffy", "Poodle", 30);
let rover = new Dog("Rover", "Mixed", 29);
let max = new Dog("Max", "Poodle", 22);
let frankie = new Dog("Frankie", "Mixed", 38);

let dogsInShow = [ fido, spot, fluffy, rover, max  ];
```

Like before, we create a bunch of dogs...

...and add them to an array. Notice this time, frankie is not included in the array. Don't worry, we'll add her later.

A Set lets you store unique values in a collection. There are no keys in a Set, just values. Unlike a Map's values, the values in a Set must be unique, but like in a Map, you can use values of any type (primitive or object).

With Maps and Sets it's best not to count on any particular ordering of keys or values in the data structure.

Let's make a Set and use it. Typically, we'll make a Set out of an array, although you can start with an empty Set and add values one at a time. We'll make a new Set from our `dogsInShow` array:

```
let dogsSet = new Set(dogsInShow);
```

We left `frankie` out of the array of `dogsInShow`, so let's add her now, using the Set's add method:

```
dogsSet.add(frankie);
```

We said every value in a Set must be unique, so let's try adding `fido` (who is already in the Set):

```
dogsSet.add(fido);
```

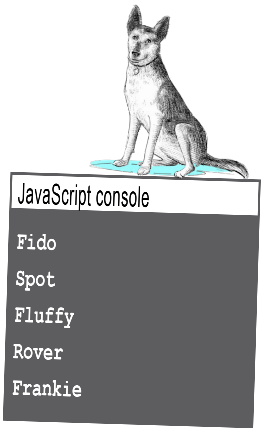
You won't see an error, but when we print out the Set, you'll see `fido` does not appear in the Set twice.

We can also delete values from a Set:

```
dogsSet.delete(max);
```

Finally, let's print the Set to the console. Like with a Map, we can use the `forEach` method. We should see `fido`, `spot`, `fluffy`, `rover`, and `frankie` (but not `max`, because we deleted `max` from the Set):

```
dogsSet.forEach(dog => console.log(dog.name));
```



```
Fido
Spot
Fluffy
Rover
Frankie
```

Sets are handy when you want to guarantee there's only one of any given value in a collection of values.

#7 Doing more with the DOM

We've touched on some of the things you can do with the DOM in this book, but there's a lot more to learn. The objects that represent the document in your page—that is, the `document` object and the various element objects—are chock-full of properties and methods you can use to interact with and manipulate your page.

You already know how to use `document.getElementById` and `document.getElementsByTagName` to get elements from the page. The `document` object has these other methods you can use to get elements, too:

Note that HTMLCollection and NodeList are "array-like" but are not arrays, so check out the documentation before you start using them.

```
document.getElementsByClassName
```

Pass this method the name of a class, and you'll get back all elements that have that class as an HTMLCollection.

```
document.getElementsByName
```

This method retrieves elements that have a name attribute with a value that matches the name you pass it.

```
document.querySelector
```

This method takes a selector (just like a CSS selector) and returns the first element that matches.

```
document.querySelectorAll
```

This method also takes a selector, but it returns all the elements that match as a NodeList.

Here's how you'd use `document.querySelector` to match a list item element with the class "song" that's nested in a `<ul>` element with the id "playlist":

```
let li = document.querySelector("#playlist .song");
```

This says match the element with the id "playlist", and then match the first element with the class song.

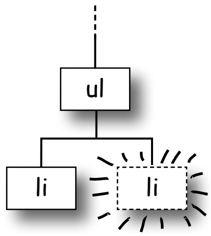
Notice how this selector is just like one you'd write in CSS?

What if you want to add new elements to your page from your code? You can use a combination of `document` object methods and element methods to do that, like this:

```
let newItem = document.createElement("li");
newItem.innerHTML = "Your Random Heart";
let ul = document.getElementById("playlist");
ul.appendChild(newItem);
```

First, we create a new <li> element and set its content to a string.

Then, we get the <ul> element we want to add the new <li> to (as a child element), and append the <li> to the <ul>.



There's a lot more you can do with the DOM using JavaScript; we encourage you to keep experimenting.

#8 The window object

You've heard of the DOM, but you should know there's also a BOM, or Browser Object Model. It's not really an official standard, but all browsers support it through the `window` object. The `window` object represents the browser window in which your script is running.

You've seen the `window` object in passing when we've used the `window.onload` property, and you'll remember we can assign an event handler to this property that is triggered when the browser has fully loaded a page.

You've also seen the `window` object when we've used the `alert` and `prompt` methods, even though it might not have been obvious. The reason it wasn't obvious is that `window` acts as the global namespace for many methods and properties that come "built in" to the `window` object, like `alert`, `prompt`, `console`, and many more. For every call we made to `alert`, we could have instead called `window.alert`, because it's the same thing.

Another place you've used the `window` object without knowing it is when you've used the `document` object to do things like get elements from the DOM with `document.getElementById`. The `document` object is a property of `window`, so we could write `window.document.getElementById(...)`. But, just like with `alert`, we don't have to, because `window` is the global object, and it is the default object for all the properties and methods we use from it.

In addition to being the global object and supplying the `onload` property and the `alert` and `prompt` methods, the `window` object supplies other interesting browser-based properties and methods. For instance, it's common to use the width and height of the browser window to tailor a web page experience to the size of the browser window.

Check out the W3C documentation* for more on the `window` object. Here are a few common methods and properties:

`window.innerWidth`
`window.innerHeight`

Use these properties to get the browser window's width and height in pixels.

`window.setTimeout()`
`window.setInterval()`

You already know these methods; they're supplied by the window object.

`window.print()`

Initiates printing the page to your printer.

`window.confirm()`

This method is similar to prompt, only it gives the user the choice of an Okay or Cancel button.

`window.history`

This property is an object containing the browsing history.

`window.location`

This property is the URL of the current page. You can also set this property to direct the browser to load a new page.

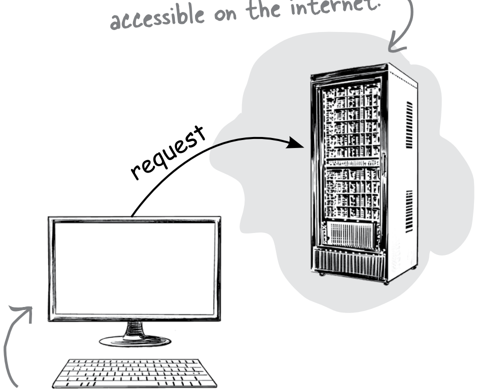
---

* https://html.spec.whatwg.org/multipage/nav-history-apis.html#the-window-object

#9 Server-side JavaScript

In this book we've focused on the browser and client-side programming, but there's a whole world of server-side programming where you can now use your JavaScript skills. Server-side programming is typically required for the kinds of web and cloud services you use on the internet. If you want to create Webville Taco's new online order system, or you think the next big idea is the anti-social network, you'll need to write code that lives and runs in the cloud (on a server on the internet).

Server-side code executes on a server on a computer accessible on the internet.



Client-side code executes on the client; that is, on your computer.

Node.js is the JavaScript server-side technology of choice these days, and it includes its own runtime environment and set of libraries (in the same way client-side JavaScript uses the browser's libraries). And like the browser, Node.js runs JavaScript in a single-threaded model where only one thread of execution can happen at a time. This leads to a programing model similar to the browser that is based on asynchronous events and an event loop.

As an example, the method below starts up a web server listening for incoming web requests. It takes a handler that is responsible for handling those requests when they occur. Notice that the convention for setting up the event handler for incoming requests is to pass an anonymous function to the `createServer` method:

```
const http = require("http");
```

Link to the library that supplies capabilities to create an HTTP server, for instance.

The http.createServer Node.js library method takes a handler in the form of an anonymous function as an argument.

```
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

The anonymous function is responsible for taking care of requests. It responds to incoming requests by sending back the string "Hello World".

Of course, there is much more to explain and to work through to understand how Node.js works. This topic requires at least an entire book of its own, but you'll also find many online tutorials, articles, and demonstrations at http://nodejs.org. Given your knowledge of objects and functions, you are well positioned to take this on.

#10 Recursion

When you give a function a name, it allows you to do something quite interesting: call that function from within the function. We call this *recursion*, or a recursive function call.

Now, why would you need such a thing? Well, some problems are inherently recursive. Here's one from mathematics: an algorithm to compute the Fibonacci number series. The Fibonacci number series is:

0, 1, 1, 2 , 3, 5, 8, 13, 21, 34, 55, 89, 144... and so on

To compute a Fibonacci number, we start by assuming:

Fibonacci of 0 is 1

Fibonacci of 1 is 1

Then, to compute any other number in the series, we simply add together the two previous numbers in the series. So:

Fibonacci of 2 is Fibonacci of 1 + Fibonacci of 0 = 2

Fibonacci of 3 is Fibonacci of 2 + Fibonacci of 1 = 3

Fibonacci of 4 is Fibonacci of 3 + Fibonacci of 2 = 5

and so on. The algorithm to compute Fibonacci numbers is inherently recursive because you compute the next number using the results of the previous two Fibonacci conmutations.

We can make a recursive function to compute Fibonacci numbers like this: to compute the Fibonacci of the number `n`, we call the Fibonacci function with the argument `n-1` and call the Fibonacci function with the argument `n-2`, and then add the results together.

Let's do that in code. We'll start by handling the cases of 0 and 1:

We start with a function that accepts n, the number in the series we're after.

```
function fibonacci(n) {
    if (n === 0) return 1;
    if (n === 1) return 1;
}
```

Then we know that if the number is either 0 or 1, we return 1. This is known as the base case of the function, because it doesn't make any recursive calls.

These are the *base cases*—that is, the cases that don't rely on previous Fibonacci numbers to compute—and it is usually good to write them first. From there, you can think like this: "To compute a Fibonacci number `n`, I return the result of adding the Fibonacci of `n-1` and the Fibonacci of `n-2`.

Let's do that…

```
function fibonacci(n) {
    if (n === 0) return 1;
    if (n === 1) return 1;
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

Now, if n isn't 0 or 1, we just need to compute the Fibonacci by adding together the Fibonacci of n–1 and n–2.

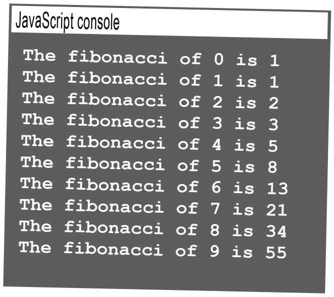
This looks a little like magic if you've never seen recursion before, but this code does compute Fibonacci numbers. Let's clean the code up a little, and test it:

Same code, just written a little better.

```
function fibonacci(n) {
    if (n === 0 || n === 1) {
        return 1;
    } else {
        return (fibonacci(n-1) + fibonacci(n-2));
    }
}
```

And some test code.

```
for (let i = 0; i < 10; i++) {
    console.log("The fibonacci of " + i + " is " + fibonacci(i));
}
```



JavaScript console

```
The fibonacci of 0 is 1
The fibonacci of 1 is 1
The fibonacci of 2 is 2
The fibonacci of 3 is 3
The fibonacci of 4 is 5
The fibonacci of 5 is 8
The fibonacci of 6 is 13
The fibonacci of 7 is 21
The fibonacci of 8 is 34
The fibonacci of 9 is 55
```

Watch it!

Make sure you have a base case.

If recursive code never reaches a base case where the computation ends, it will run forever, like an infinite loop. In other words, the function will continue calling itself over and over, consuming resources until your browser can't take it anymore. So if you write recursive code and your page isn't responding, make sure you're getting to the base case.

Index

Symbols

` (back tick mark) 307–308
++ (post-increment) operator 153–156, 370
+ (plus sign)
  as arithmetic operator 296–297
  in concatenating strings 17, 149, 368
<= (less than or equal to) operator 59, 291
< (less than) operator 59
=== (strict equality) operator 285, 290–295
  versus == operator 299
  as comparison operator 59, 324
== (equality) operator 285–295
  versus === operator 299
  as comparison operator 18, 59, 324
!== (strict not equal to) operator 291
=> (arrow function syntax) 458
= (assignment) operator 11, 18, 295
!= (not equal to) operator 18, 59
>= (greater than or equal to) operator 18, 59, 291, 462
> (greater than) operator 18, 59
|| (OR) operator 58, 59, 66–67, 78
$ (dollar sign) character 15
${ }, for enclosing expressions 308
0 (zero)
  as falsey value 302
  dividing by 284
&& (AND) operator 59, 66–67, 78
* (asterisk) as multiplication operator 17, 296
\ backslash (escape) character 565
, (comma) character 187, 579
{ } (curly braces)
  arrow functions 461
  code blocks 19, 106, 147
  function body 87
  object declaration 187
  object literals 544
  template literals 308
. (dot notation) 189–191, 194, 210–215, 217, 230
// (forward slashes) 15
/ (forward slash) operator 17, 296
-Infinity value 284
- (minus sign) operator 296–297
! (NOT) operator 59
() (parentheses)
  arrow functions 474
  assigning function values 259
  calling functions 72
  declaring functions 101
  declaring methods 218
  and left-to-right associativity 297
-- (post-decrement) operator 153
“ ” (quotation marks, double) 15, 308
. . . (rest operator) 533
; (semicolon) character 11, 15, 146, 459
. . . (spread operator) 494–496
[ ] (square brackets)
  accessing array items 133, 217
  creating arrays 132, 134
  updating array items 135
  accessing object properties 217
_ (underscore) character 15

A

accessor properties 575–577
action attribute 342
addFuel() method 219–220
alert() function 27–28, 50, 63–64
AND (&&) operator 59, 66–67, 78
anonymous functions 485, 497
  creating and using 454–458, 486
  as right here right now functions 473

applications, JavaScript in 5
arguments 89, 92–99
  adding to object literal 572
  creating object from class 549
  default parameters 492–494
  in event handlers 414
  order of 101
  passing by value 200
array literals 132, 158–159
arrays 25, 131–162
  accessing items 133, 135
  adding items 158–159
  basic car simulator 561
  creating 132–134, 158–159
  destructuring 598
  empty 140, 158, 352
  filter() method 471–473, 477–479, 488
  forEach() method 471, 480–482
  indexes 134, 140
  indexOf() method 357
  item types 140
  iterating over 144–157, 160–162
  last item 140
  length property 136–139, 175
  map() method 471, 474, 477–479
  maximum length 140
  nesting objects and arrays 362
  order of items 140
  parallel 169–172, 179
  sort() method 465–470, 487
  sparse 158–159
  undefined items 159, 278
  updating items 135
array sort method 465–470, 487
arrow functions 458–465, 473, 485, 497, 501
arrow function syntax (=>) 458
assignment (=) operator 11, 18, 295
asterisk (*) operator 17, 296
asynchronous code 373–376, 403, 429, 597
asynchronous operations 597
attributes
  getting 266
  setting 264–267, 273

B

background property 340
backslash (\) character 565
back tick mark (`) 307–308
bake() function 99–101
bark() function 87–90
bark() method 554
base cases 606–607
Battleship game, advanced 331–396
  cheating at 384
  controller object 343, 363–371, 375
  CSS 338–342
  event handlers 373–376
  HTML 332–337
  model object 343, 350–362, 379–380
  randomizing ship locations 377–383
  toolkit 333
  view object 343–349, 361, 391
Battleship game, simplified 48–68, 70–77, 80–81
  code for 64
  communicating with users 62–63
  design 48–50
  finding errors 72–73
  hit detection 60–61
  HTML 53–58
  pseudocode 51–56
  quality assurance 65, 73
  randomizing ship locations 70–72
  user input 57–59
behavior 188, 219–221, 224. *See also* methods
BigInt 599
block scope 106–107
blocks of code 19, 25
  in for loops 147
  scope 106–107
  and shadowing variables 114
boardSize property 355, 362
<body> element 4
  adding code to 34
  in DOM 244–245
  replacing content 255
body, function 87–90

BOM (Browser Object Model) 604

boolean expressions 17–18
  in if statements 24
  syntax 15
  truthy and falsey values 301–303

boolean operators 59
  complex statements 66–68, 78–79
  verbose conditionals using 69

boolean values 11, 25
  as objects 306
  comparing to other types 287–288, 291–292

bracket ([ ]) notation 217. *See also* square brackets ([ ])

Browser Object Model (BOM) 604

browsers
  building DOM 243
  default parameters 493, 535
  event handling 399, 419–420, 424. *See also* event handlers
  loading and executing code 3, 33, 498–499
  opening console in 30

browser wars 6

Bubbles-R-Us project
  bubble score report 149–150, 154–157, 160–162
  data 132
  order of items 140
  parallel arrays 169–172
  refactoring 164–168

built-in functions 75

built-in objects 223–224

C

calculateArea() function 100

callback hell 597

callbacks. *See* event handlers

camel case in multiword variable names 15

Car class 560, 563–565, 571–573

car object 204–205

Cascading Style Sheets (CSS)
  adding whitespace with 410
  background property 340–341, 390
  Battleship game, advanced 333
  versus JavaScript 9–10
  positioning elements 338–340, 342
  selectors 253, 255
  src property 406–407
  in web page ecosystem 2–3

case sensitivity 14–15

chaining 359, 362, 477–479, 488

character strings, quoting 15

character type, absence of 309

charAt() method 309, 314

chevy object 217

classes 547–553, 556
  adding properties 567
  creating objects with 549–550
  declaring 548–549
  extending 563–565, 568, 570
  and methods 564
  naming 553
  versus object literals 555
  and object-oriented inheritance 560–570
  static properties and methods 582–585

classes (CSS)
  labeling HTML elements with 253
  retrieving elements by 255
  setting 265

class keyword 548

client-side code 605

closures 516–532
  creating 516–519
  magic counter implementation 521–525
  makeTimer 523–525
  onlyOnceMaker() function 530–533
  shared environment 525

code
  asynchronous 403
  compiling 5
  finding errors in 27, 29–30, 72
  loading into browser 33
  refactoring 163–168

code blocks 19, 25
  in for loops 147
  scope 106–107
  and shadowing variables 114

code files 34, 35–36

code hygiene 117

code reuse 84–87, 216

collision() method 379, 383

comma (,) character 187, 579

comments 15

communication with users. *See* users, communicating with

compare() function 466–467

compareSold() function 465, 468–469

comparison operators 59

compiling code 5

concatenating strings 17, 139, 296–297, 368

conditionals
  || (OR) operator 58
  && (AND) operator 59, 66–67, 78
  boolean expressions in 24
  complex 66–68, 78
  falsey and truthy values in 301–303
  variables or strings in 25
  verbose versus succinct 69
  in while loops 19–23

confirm() method 604

console
  Console object 223–224
  finding errors with 72
  opening in browser 30, 150

console.log 29–33
  communicating with users using 27–29
  displaying objects 193
  formatting output with function 469
  passing arrays to 311
  passing innerHTML property to 252

constants 13, 101

const keyword 13, 41, 117

constructor() function 548, 552, 572–573

constructors, object 546–592
  accessor properties 575–577
  classes. *See* classes
  creating objects with 558–562
  finding errors in code 552, 588
  getters and setters 576–580
  versus literals 553, 571–573
  methods in 554–555, 557, 590
  naming conventions 553
  parameter names in 553
  static properties and methods 582–585
  workings of 551

controller object, advanced Battleship game 343, 363–371, 375

convention, creating objects by 545

cookieAlarm() function 457–458

countDogsByBreed() function 601–602

counter property 581, 582–584

counters
  implementing with closures 521–525
  initializing and incrementing 146, 152

count variable 521

crack the code challenge 225, 240–242

createFleet() function 594–595

createFortuneCookies() function 539

createGreeting() function 516–517

createServer() method 605

CSS. *See* Cascading Style Sheets (CSS)

curly braces ({ })
  arrow functions 461
  code blocks 19, 106, 147
  function body 87
  object declaration 187
  object literals 544
  template literals 308

D

data properties 575, 577

data structures. *See also* arrays; objects
  in advanced Battleship game 352–354
  Map 600–602
  Set 602

Date object 223

default parameters 492–495

default values 598

delete expression 192

destructuring assignment 598

displayHit() method 345–349

displayMessage() method 345–349

displayMiss() method 345–349

<div> element 338–339

doCount() function 522, 523

document.getElementById() method. *See* getElementById() method

document.getElementsByClassName() method 255, 603

document.getElementsByName() method 603

document object 27–28, 223, 245, 250, 603

Document Object Model (DOM) 239–273, 603
  crack the code challenge 225, 240–242
  creating 243–247, 271
  event handlers. *See* event handlers
  functionality 268
  waiting for page to load 258–263
  window.onload property. *See* window.onload property

document.querySelectorAll() method 255, 603

document.querySelector() method 255, 603

document.write() method 27–28

Dog class 551–552, 554

dollar sign ($) character 15

DOM. *See* Document Object Model (DOM)

doneMessage variable 526

dot notation (.) 189–191, 194, 210–215, 217, 230

do while loops 379

dragstart event 435

drive() method 206–211, 220, 567, 569

drop event 435

E

ECMAScript 6

element object 251, 255, 407

elements
  adding new 603
  attributes, getting 266
  attributes, setting 264–267, 273
  innerHTML property. *See* innerHTML property
  positioning with CSS 338–342
  removing 268
  retrieving by class 255
  retrieving by id. *See* getElementById() method
  retrieving by tag name 255
  retrieving with querySelectorAll() method 255, 603
  retrieving with querySelector() method 603

else statement, as catch-all 25

encapsulatation 193

equality (==) operator 285–295
  versus === operator 299
  as comparison operator 18, 59

errors in code, finding 27, 29–30, 72

escape (\) character 565

event handlers 260–263, 399–417, 423–433
  asynchronous coding 373–376
  creating 400–401, 404–408
  and event objects 414–418, 439
  and event queue 420
  and flow of execution 407
  mouseover event 430–433
  and Promises 597
  reusing 409–413
  server-side JavaScript 605
  and setTimeout() method 423–429

event objects 414–418, 429, 439

event queue 419–422, 440

events 397–444
  event queue 419–422, 440
  exercises 436–438
  handling. *See* event handlers
  kinds of 262, 419, 423, 429, 435
  organizing code around 403

examples
  files for xxxi
  HTML wrapper for xxxii

exercises
  === (strict equality) operator 290, 324
  == (equality) operator 290, 324
  = (assignment) operator 290, 324
  || (OR) operator 60, 81
  accessing array items 136
  anonymous functions 454, 456–457, 485, 486
  arguments 94–95, 97, 111, 125
  array methods 476, 479
  arrays 133, 136, 142, 145, 148, 156, 160–161, 171, 174–180
  arrow functions 460, 485

Battleship game, advanced 332, 341, 344, 351, 353–354, 360, 371–375, 378, 383, 387, 389–395
Battleship game, simplified 52, 53, 56, 60, 63, 74, 77, 80–81
boolean expressions 67–69, 78–79
bubble score report 142–143, 156, 160–161, 167, 171
chaining 479, 488
classes 555, 566, 588
closures 519, 524, 528, 530–533, 539, 540, 541
code analysis 84, 122
communicating with users 45
compare functions 467, 470, 485, 487
comparing JavaScript output 28, 45
constructors 587, 592
controller object 396
creating objects with classes 550
CSS 264, 273, 332, 341, 390
default parameters 493, 535
DOM 225, 246, 256–257, 263, 264, 270–274
dot notation 191, 230
errors in constructor code 552, 588
event handlers 402, 409, 411, 418
event object 416, 439
event queue 421–422, 440–441
events 398, 408, 423, 426, 430–431, 441–443
execution flow 408, 438
expressions 18, 43
falsey and truthy values 303, 326
filter() method 476, 488
for loops 148, 156, 160–161, 177–178, 383
function declarations 448
function expressions 448, 484
functions 90–91, 94–95, 97, 111, 116, 118–119, 121–130, 199, 202, 432–433, 454–456, 460, 464, 467, 470, 476, 479, 484–489
functions as values 451, 484
general 7–8, 40–41, 45, 76, 82
getElementById() method 256
HTML 332
instanceof operator 318, 328
isNaN() function 315
length property 175
loops 26, 56
method shorthand 218, 234–235
methods in constructors 557, 590
mouseover event 430
nested functions 505
nested loops 383
null values 280, 323
object creation 186, 191, 229–230
object literals 573
object references 300, 327
objects 183–184, 211, 213–215, 218, 221, 227–237
onmouseover property 442
onresize property 431, 443
parallel arrays 171, 179
parameters 94–95, 97, 111, 125, 464
passing objects to functions 199, 202–207, 231–232
properties 183–184, 186, 228, 229
pseudocode 52, 63, 77, 80, 171, 179
push() method 161
quality assurance 74–75
reduce() method 476, 488
refactoring code 167–168
rest parameters 494, 535
scope 111, 125, 511, 534, 538, 542
setInterval() method 426, 441
setTimeout() method 423
split() method 317
state 221, 236–237
strings 284, 304, 315, 317
syntax 16, 42, 496, 536
this keyword 211, 213–215, 228, 233–234
two passes over code 502, 537
type conversion 297, 325
typeof operator 279–280, 322–323
types 277, 284, 290, 300, 303–304, 318, 320–321, 323–329
users, communicating with 28
variable declaration 12, 13, 41
view object 344, 391
while loops 26, 44, 145, 176
window.onload property 263, 272

export keyword 594–595

expressions 17–18, 43
  chaining 359
  function expressions 446–452
  passing as arguments 93
  in template literals 308

extending classes 562–564

extends keyword 563, 568, 570

F

false (boolean value) 15. *See also* boolean values
falsey and truthy values 301–303, 326
fetch() function 597
fiat object 206–212, 219–222
Fibonacci number series 606–607
filter() method 471–473, 477–479, 488
fire() method 356–358, 360, 370
first class functions 453, 462, 465
first class values 453
floating point number, maximum 599
flow of execution, tracing 100
fly() function 446–450
forEach() method 471, 480–482, 601
for in iterator 217
for loops 146–151, 160–162, 177–178
  with arrays. *See* forEach() method
  Battleship game, advanced 357, 380, 382
  nested 383
  with parallel arrays 170–172
  post-decrement (--) operator 153
  post-increment (++) operator 153–156
  scope 147, 159
  versus while loops 151–152
<form> element
  action attribute in 342
  adding inputs to 337
  event handlers with 373–376
  positioning with CSS 339
fortuneCookie() function 510–511
forward slashes (//) 15
forward slash (/) operator 17, 296
free variables 524
function declarations 446–450, 497–501
function echo 494, 535
function expressions 446–452, 462, 484, 497, 500
function references 446–452
functions 87–130, 445–489, 491–542
  anonymous. *See* anonymous functions
  arguments. *See* arguments
  arrow functions 458–465, 473, 485, 497, 501
  benefits of 83, 92
  built-in 75
  calling (invoking) 89–90, 93–95, 214
  and closures 516–532
  constructor. *See* constructors, object
  environments for 524, 527
  function declarations 446–450, 497–501
  function expressions 446–452, 462, 484, 497, 500
  higher-order 471–482, 489
  hoisting 498–502
  location in file 115, 117
  versus methods 206, 214
  naming 101, 214, 449
  nesting 504–509
  parameters. *See* parameters
  parentheses in. *See* parentheses ()
  passing objects to 196–205
  passing to array sort method 465–470
  passing to other functions 425, 452, 471–482
  as properties 206
  properties in objects 200
  recursion 606–607
  refactoring code into 163–168
  returning a function from a function 514, 516
  returning objects from 205
  return statement 99–101, 461
  right here right now functions 473
  scope. *See* scope
  scoping environment for evaluation 512
  syntax 87–91, 101, 492
  tracing 100
  as values 447–454, 484

G

generateShipLocations() method 379, 382, 384
generateShip() method 379–380, 382
getAttribute() method 266, 407
getBestResults() function 167–168
getCounter() method 582–584
getElementById() method 240–242, 247–251, 253, 256–260
  Battleship game, advanced 374
  nonexistent elements 255–258, 266, 280–281

getElementsByClassName() method 255, 603
getElementsByName() method 603
getElementsByTagName() method 255, 412–413
getFortune() function 510–511
get keyword 575
get km() method 576–580
getSize() method 557, 590
getter methods 575–577
global scope. *See* scope
globalVariable 507
global variables 103–114, 117
greater than (>) operator 18, 59
greater than or equal to (>=) operator 18, 59, 291, 462
guesses property 363, 370
guesses variable 58

H

hail() method 564
handleFireButton() function 374–375
handleKeyPress() function 376
handler() function 455–456, 459
handlers. *See* event handlers
hardcoding 355
<head> element 4, 34, 244–245
higher-order functions 471–482, 489–490
history property 604
hit class 340–341
hits property 352
hoisting functions 498–502
hoisting variables 114
HTML
  Battleship game, advanced 333–338
  Battleship game, simplified 53
  creating DOM from 243–246
  creating tables 336
  editors 33
  id attribute 253
  in web page ecosystem 2–3
  versus JavaScript 9–10, 243, 247
HTMLCollection 413
<html> element 4, 244–246
HTML wrapper, for examples xxxii
http.createServer() method 605

I

i
  in for loops 147
  as variable name 104
id attribute/property 253
  in advanced Battleship game 336
  of element object 407
  in image guessing game 414, 417
  numbers as 342
identity operator. *See* === (strict equality) operator
ids, generating unique 599
if/else statements 24–25
if statements 24–25
image guessing game 404–406, 429, 528, 540
images, adding to <td> element 340
imaginary numbers 282
<img> element
  retrieving all 412–413
  setting src property 406–407
includes() method 312–313, 383
indexes 133–134, 140
indexOf() method 310, 357, 359, 366, 383
infinite loops 59
Infinity value 284
inheritance. *See* object-oriented inheritance
init() function 259–261, 374
innerFunction() function 504–506, 508
inner functions 504–509
innerHeight property 604
innerHTML property
  accessing text with 240–241

  - in advanced Battleship game 345–347
  - updating text with 249, 251–252, 254–255, 257

innermostFunction() 507

innermostVariable, nested functions 507

innerVariable, nested functions 504–506, 507

innerWidth property 604

<input> element 337, 339

instanceof operator 318, 328

integer number, maximum 599

interpreted languages 5

isCola() function 472–473

isNaN() function 283, 315, 316, 367

isSunk() method 360

iteration. *See also* loops
  - over arrays 144–157, 160–162
  - through objects 217

J

Java 5

JavaScript
  - adding to web pages 4–5, 32–37
  - versus CSS 9–10
  - history of 5–6, 38–39
  - how to learn xxix–xxxii
  - versus HTML 9–10, 243, 247
  - in applications 5
  - in external files 35–37
  - in web page ecosystem 2–3
  - versus Java 5
  - in <script> element 4–5
  - server-side 605

JScript 6

JSON (JavaScript Object Notation) 596

JSON object 223

K

keypress event 413, 435

keys
  - for Map data structure 600
  - generating unique 599

keywords 14

L

lastIndexOf() method 313

left property 338–339

left-to-right associativity 297

length property
  - of arrays 136–139, 175
  - of strings 309, 314

less than (<) operator 59

less than or equal to (<=) operator 59, 291

let keyword 11–13, 41, 117

letters, converting to numbers 342, 364, 366

lexical scope 507, 513. *See also* scope

listeners. *See* event handlers

literals
  - array 132, 158–159
  - object. *See* object literals
  - template 307–308, 565

LiveScript 6

load event 435. *See also* window.onload property

local scope. *See* scope

local variables 103–114, 117, 513

location property, window object 604

locations property, advanced Battleship game 352

logical operators 59

loop blocks 147

loops 19–23
  - do while 379
  - for. *See* for loops
  - infinite 57, 59
  - nested 383
  - pseudocode 55–56
  - while. *See* while loops

loseWeight() function 201

M

makeCar() function 204–205

makeCounter() function 521, 522

makePhrases() function 137–139

makeTimer() function 523–527

Map data structure 600–602

map() method 471, 474, 477–479

margin property 410

match() method 313

Math.floor() method 71, 139, 380–382, 474

Math object 223

Math.random() method 70–71, 139, 380–382, 448

metacognition xxvii

<meta> element 245–246

methods
  adding to objects 206–210
  calling 214
  in constructors 554–555, 557, 590
  creating objects by convention 545
  declaring with shorthand 218–219, 234–235
  versus directly changing properties 208
  versus functions 206, 214
  getter and setter 575–577
  overriding 567, 570
  passing arguments to 214
  as properties 224
  with properties 545, 551, 561
  returning values from 214
  static 582–585
  and this reference 553

method shorthand 218–219

Microsoft, introduction of JScript 6

minus sign (-) operator 296–297

miss class 340–341

model object, advanced Battleship game 343, 350–362, 361, 379–380

model.ships property 383

modules 594–595

mousemove event 421–423, 435

mouseout event 435

mouseover event 430–433, 435

N

naming
  classes 553
  functions 87, 101, 214, 449. *See also* anonymous functions
  and shadowing variables 110, 114
  variables 101, 104

NaN (Not a Number) values 282–284, 291, 302

nested blocks 508

nested functions 504–509

Netscape 6

new keyword 549

new operator 549, 551

Node.js 605

not equal to (!=) operator 18, 59

NOT (!) operator 59

null values
  assigning 280
  comparing to undefined 288
  as falsey 302
  returned by getElementById() method 255
  returned by prompt() function 59
  testing for 281
  typeof 284
  versus undefined 323

Number() function 297

Number.MAX_SAFE_INTEGER 599

Number.MAX_VALUE 599

numbers
  BigInt 599
  comparing to strings 59, 285–288, 291
  concatenating strings to 296–297
  generating random 70–71
  and NaN (Not a Number) values 283–284, 291, 302
  as objects 306
  unrepresentable 282–284

numShips property 355

O

object literals. *See also* objects
  versus classes 553, 555
  creating for car classes 573
  creating objects 544
  disadvantages of 547
  getters and setters with 579–580
  passing values to constructors 571–573
object-oriented inheritance 559–570
  adding methods 564
  extending classes with 562, 570
  overriding methods 567–568
object-oriented (OO) programming 188
objects 181–224, 226–237
  adding methods to 206–210, 214, 218–219
  in advanced Battleship game. *See* Battleship game, advanced
  behavior 219–221, 224
  benefits of 346, 349
  booleans as 306
  built-in 223–224
  class versus literal creation of 553
  constructors. *See* constructors, object
  by convention 545
  creating 185–187, 193, 229–230, 544, 571–573
  destructuring assignment 598
  equality of 298–300, 327
  functions. *See* functions
  iterating through 217
  versus Map data structure 600
  methods. *See* methods
  nesting in arrays 362
  numbers as 306
  object-oriented (OO) programming 188
  passing to functions 196–205, 231–232
  properties. *See* properties
  references 194–195
  returning from functions 205
  state 219–221, 224, 236–237
  string objects 304–306
  testing for existence 281
  this keyword 210–215, 228, 233–234
  turning into strings 596
  versus primitive types 276
object variables 194
onceOnlyMaker() function 532
onclick event handler 374, 405–407, 409–414
onclick property 406, 435
onload event handler. *See* window.onload property
onload property. *See* window.onload property
onlyOnceMaker() function 541
onmousemove property 421–422
onmouseout property 430
onmouseover property 430, 442
onresize property 431, 443
OO (object-oriented) programming 188
operands 279
OR (||) operator 58–59, 66–67, 78
outerFunction() function 504, 507, 508
outer functions 504–509
outerHTML property 255
outerVariable, nested functions 504–507
overriding methods 567, 570

P

pageLoadedHandler() function 401–402
parallel arrays 169, 179
parameterization 88, 471
parameters 87–89, 93–97, 101–107
  constructor 548, 553, 571–572
  default 492–495
  and object literals 572
  and passing functions to other functions 471
  rest 494
  shadowing global variables 114
params parameter 574
parentheses ()
  arrow functions 474
  assigning function values 259
  calling functions 72
  declaring functions 101
  declaring methods 218
  and left-to-right associativity 297

parseGuess() function 365–368

pass-by-value 96–97, 200

pause event 435

pay() method 564

Phrase-o-Matic app 137–139

play event 435

playTurn() function 106

plus sign (+)
  as arithmetic operator 296–297
  in concatenating strings 17, 149, 368

positioning, CSS 338–342

position, relative versus absolute 338, 342

post-decrement (--) operator 153

post-increment (++) operator 153–156, 370

prequal() function 196–199

primitive types 25
  strings as objects versus 304–306
  supplied by JavaScript 276

primitive values 195

primitive variables 194

printAndGetHighScore() function 165–166

print() method 604

procedural design 343

procedural programming 188

processGuess() method 363–364, 372

programming languages, versus HTML and CSS 9

Promises 597

prompt() function 50, 57–58, 59

properties
  accessing 189–193, 210–215, 217, 600
  accessor 575–577
  adding existing 193
  adding to constructor 567
  arguments in object literals 572
  changing in functions 200–201
  creating 182–184, 228, 229
  creating objects without 193
  deleting 192
  functions as 206
  iterating through, in objects 217
  maximum allowable 193
  with methods 545, 551, 561
  methods as 224
  and object state 219
  overriding conditions 570
  for state and behavior 224
  static 582–585
  syntax 187, 193
  undefined 193, 278

pseudocode 51–56, 179

push() method 158–159

Q

quack() function 446–450

quality assurance (QA) 65, 73–74

querySelectorAll() method 255, 603

querySelector() method 255, 603

quotation marks, double (“ ”) 15, 308

R

rand() function 510–511

random number generation 70–71

reblur() handler 427–429

recursion 606–607

reduce() method 471, 475–479, 488

refactoring code 163–168

reference errors 108, 147, 209–210

references and reference variables 194–195, 200–205, 327
  chaining 359, 362
  function references 446–452
  and object equality 298–299
  passing references to functions 425
  reference errors 209–210

RegExp object 223

replaceAll() method 312–313

replace() method 312–313, 317

reserved words 14

resetCounter() method 584

resize event 435

rest operator (. . .) 533

rest parameters 494

rest values 598

return statement 99–101

right here right now functions 473

RocketCar class 567–570

RocketCar constructor 566

S

sample files xxxii

scope 102–103, 105–115
  blocks as source of variable definition 512
  closures 516–532
  in code blocks 106–107
  counter variables in loops 147, 159
  encapsulation with nested functions 510–511
  and event handlers 407
  taking functions beyond global 504
  global and local variables 103–114, 117, 513
  kinds of 107
  lexical versus global 513
  nesting functions 504–509
  returning a function from a function 514
  shadowing global variables 110
  and variable life 109
  variables used prior to declaration 108, 114

scoping environment 512, 513, 524

<script> element 4, 32–33, 37–38
  in DOM 245–246
  in head element 34
  location in file 258

scripting languages 5

semicolon (;) character 11, 15, 146, 459

server-side JavaScript 605

server-side programming 605

setAttribute() method 265, 347–349, 407

Set data structure 602

setInterval() method 426, 441, 604

set keyword 577

set km() method 577

set() method 600–601

setter methods 577–578

setTimeout() method 423–429, 457–458, 527, 604

shadowing variables 110, 114

shipLength property 355

ships property 355

showAnswer() function 406–407, 409–411, 414–415, 429

slashes (//) 15

slash (/) operator 17, 296

slice() method 313

sort functions 452

sort() method 465–470, 487

sparse arrays 158–159

split() method 311

spread operator (. . .) 495, 496

square brackets ([ ])
  accessing array items 133, 217
  creating arrays 132, 134
  updating array items 135
  accessing object properties 217

src attribute, in <script> tag 37–38

src property 406–407, 417

started property 207–210

start() method 206–207

state 188, 219–221, 224, 236–237

statements 10–11, 15, 16. *See also* function declarations

static keyword 582

static properties and methods 582–585

stop() method 206

strict equality (===) operator 285, 290–295
  versus == operator 299
  as comparison operator 59, 324

strict not equal to (!==) operator 291

strings 304–318
  comparing to numbers 59, 285–288, 291
  comparing with < and > operators 291
  concatenating 17, 139, 296–297, 368

  in conditionals 25, 302
  empty 302
  and isNaN() function 284
  methods and properties 306, 309–317
  as primitives and objects 304–306
  reducing array of 476–477
  splitting across lines 149
  template literals 307–308
  turning into objects 596
subclass 570
substring() method 311–312, 313, 315, 316
sum() function 495
Sun Microsystems 6
superclass 568, 570
super.drive() method 568, 569
super() function 563–565, 568, 584
Symbols 599
syntax 15–16, 42, 101, 492

T

<table> element 336, 339
tag names, retrieving elements by 255
target property 414–415
Taxi class 563–565, 594–595
<td> element 336, 339, 340–341, 347–348
template literals 307–308, 565
text editors 33
this keyword 210–215, 228, 233
this reference, objects from classes 551–553
thrust() method 569
thrustOn property, adding to constructor 567
timerHandler() handler 423–425
timers. *See* setTimeout() method
toFixed() method 565
toLowerCase() method 313
top property 338–339
toString() method 557, 590
touchend event 435
touchstart event 435
toUpperCase() method 313
trim() method 313
true and false (boolean values) 15, 17–18, 301–303, 326
truthy and falsey values 301–303, 326
type attribute, in <script> tag 37
type conversion 286–289, 297
  and equality (==) operator 325
  concatenating numbers to strings 296–297
  converting strings to numbers 296–297, 367
  temporary, for strings 305
typeof operator 279, 318, 322–323, 328, 448
types 275–329
  and == (equality) operator 285–295
  absence of character type 309
  BigInt 599
  conversion 286–289, 296–297, 325, 367
  falsey and truthy values 301–303, 326
  instanceof operator 318, 328
  NaN (Not a Number) 282–284
  null values 59, 280–281, 284, 323
  and object references 298–300, 327
  primitive 276
  strings. *See* strings
  typeof operator 279, 318, 322–323, 328
  undefined values. *See* undefined values
TypeScript 5

U

Uncaught ReferenceError 108, 147, 595
Uncaught TypeError 258
undefined values 278
  in arrays 159
  as default value of variables 54
  as falsey 302
  functions without return statements 101
  versus null 280, 323
underscore (_) character 15
Unicode values 291
unload event 435
users, communicating with
  alert() function 27–28, 50
  Battleship game, simplified 62
  with console.log 27–29

  - <div> element 337
  - through DOM 27
  - prompt() function 50, 57–58

V

values
  - assigning 11
  - equality and 286, 290–295
  - first class 453
  - function. *See also* first class functions; function declarations; function expressions
  - functions as 447–453
  - generating unique 599
  - NaN (Not a Number) 282–284, 291, 302
  - null. *See* null values
  - operators and 296
  - passing to functions 93, 96–97
  - primitive 195
  - strings 301–307
  - truthy and falsey 301–303
  - and type conversion 286–289
  - undefined 278

variables
  - assigning functions to 497
  - in conditionals 25
  - declaring 11–13, 41
  - destructuring assignment 598
  - free 524
  - global. *See* global variables
  - lexical scope of 507
  - life 109
  - local. *See* local variables
  - naming 14–15, 101, 104
  - object 194
  - passing as arguments 93
  - primitive 194
  - reference 194–195
  - scope 102–103, 513
  - shadowing 110, 114
  - undefined 54, 278
  - used prior to declaration 108, 114

var keyword 11, 14–15, 117

view object, advanced Battleship game 343–349, 361, 391

W

web browsers. *See* browsers

web page ecosystem 2–3

web pages
  - adding background image and color 335
  - adding code to 4–5, 32–37
  - creating and loading in browser 33
  - reloading, effect on code 114
  - waiting to fully load 258–263

while loops 19–23, 26, 44
  - versus for loops 151–152
  - iterating over arrays 144–145

white space in JavaScript code 15

Widget class 582–583

window object 223, 259, 426, 604

window.onload property 259–262, 272
  - in advanced Battleship game 374
  - event handler for 401, 454–455
  - omitting window on 426

write() method 27–28

Z

zero (0)
  - as falsey value 302
  - dividing by 284

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2023 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 175_8 x 9.25